libc.a reference

Introduction

The standard C library, `libc.a`, is automatically linked into your programs by the `gcc` control program. It provides many of the functions that are normally associated with C programs. This document gives the proper usage information about each of the functions and variables found in `libc.a`.

For each function or variable that the library provides, the definition of that symbol will include information on which header files to include in your source to obtain prototypes and type definitions relevant to the use of that symbol.

Note that many of the functions in `libm.a` (the math library) are defined in `math.h` but are not present in libc.a. Some are, which may get confusing, but the rule of thumb is this---the C library contains those functions that ANSI dictates must exist, so that you don't need the `-lm` if you only use ANSI functions. In contrast, `libm.a` contains more functions and supports additional functionality such as the `matherr` call-back and compliance to several alternative standards of behavior in case of FP errors. See libm, for more details.

Debugging support functions are in the library `libdbg.a`; link your program with `-ldbg` to use them.

Functional Categories

Alphabetical List

_8087

Syntax

```
#include <dos.h>

extern int _8087;
```

Description

This variable is provided for compatibility with other DOS compilers. It contains `3` if a numeric coprocessor is installed, otherwise `0`. If the environment variable `387` is set to either `y` or `n`, the value of `_8087` reflects the override (i.e., `_8087` is unconditionally assigned the value 3 if `387` is set to `y`, 0 if it is set to `n`).

Portability

{ANSI/ISO C

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

a64l

Syntax

```
#include <stdlib.h>

long a64l(const char *string);
```

Description

This function takes a pointer to a radix-64 representation, with the first digit the least significant, and returns the corresponding `long` value.

If string contains more than six characters, only the first six are used. If the first six characters of string contain a null terminator, only those characters before the null terminator are used. `a64l` will scan the string from left to right, with the least significant digit on the left, decoding each character as a 6-bit radix-64 number.

The radix-64 representation used by this function is described in the documentation for the `l64a` function (See l64a).

Return Value

Returns the `long` value resulting from the conversion of the contents of string, or `0L` if string is `NULL`, points to an empty string, or points to an invalid string (i.e. one not generated by a previous call to `l64a`). If the result would overflow a `long`, the conversion of `/2BIG/` (`1144341633L`) is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. This function is new to the Posix 1003.1-200x draft

abort

Syntax

```
#include <stdlib.h>

void abort(void);
```

Description

When you call `abort`, the message "Abort!" is printed on stdout and the program is aborted by calling `raise (SIGABRT)` (See signal, SIGABRT). By default, this causes the CPU registers and the call frame stack dump to be printed, and the program then exits with an exit code of -1 (255). If the `SIGABRT` signal is caught by a handler that returns, the program exits with an exit code of 1.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if ((q = malloc(100)) == NULL)
abort();
```

abs

Syntax

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

```
int abs(int value);
```

Return Value

The absolute value of `value` is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int sq = 7;
sq = sq * abs(sq) + 1;
```

access

Syntax

```
#include <unistd.h>

int access(const char *filename, int flags);
```

Description

This function determines what kind of access modes a given file allows. The parameter flags is the logical `or` of one or more of the following flags:

`R_OK`
: Request if the file is readable. Since all files are readable under MS-DOS, this access mode always exists.

`W_OK`
: Request if the file is writable.

`X_OK`
: Request if the file is executable.

`F_OK`
: Request if the file exists.

`D_OK`
: Request if the file is really a directory.

Return Value

Zero if the requested access mode is allowed, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
if (access("file.ext", W_OK))
return ERROR_CANNOT_WRITE;
open("file.ext", O_RDWR);
```

acos

Syntax

```
#include <math.h>

double acos(double x);
```

Description

This function returns the angle in the range `[0..Pi]` radians whose cosine is x. If the absolute value of x is greater than 1, a domain error occurs, and `errno` is set to `EDOM`.

Return Value

The arc cosine, in radians, of x. If the absolute value of x is greater than 1, the function returns a `NaN`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

acosh

Syntax

```
#include <math.h>

double acosh(double x);
```

Description

This function returns the inverse hyperbolic cosine of x.

Return Value

The inverse hyperbolic cosine of x. If the value of x is less than 1, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

addmntent

Syntax

```
#include <mntent.h>

int addmntent(FILE *filep, const struct mntent *mnt);
```

Description

This function is a no-op for MS-DOS, but is provided to assist in Unix ports. See getmntent.

Return Value

This function always returns nonzero to signify an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

alarm

Syntax

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

Description

This function causes the signal `SIGALRM` to be raised in seconds seconds. A value of zero for seconds cancels any pending alarm. If an alarm has previously been set, the new alarm delay will supercede the prior call.

Note that signals in DJGPP are deferred when the program is inside a real-mode (e.g., DOS) call or isn't touching its data; see See signal, for more details.

A misfeature of Windows 9X prevents the timer tick interrupt from being delivered to programs that are in the background (i.e. don't have the focus), even though the program itself might continue to run, if you uncheck the Background: Always suspend property in the Property Sheets. Therefore, `alarm` will not work in background programs on Windows 9X.

Return Value

The number of seconds remaining on the timer (i.e. always seconds).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
signal(SIGALRM,my_alarm_routine);
alarm(5);
```

alloca

Syntax

```
#include <stdlib.h>

void *alloca(size_t _size)
```

Description

Allocate memory that will be automatically released when the current procedure exits. Note that, when compiling with gcc, alloca is a built-in function and not a library call.

Return Value

A pointer to the memory, else NULL.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
q = alloca(strlen(x)+1);
strcpy(q, x);
```

asctime

Syntax

```
#include <time.h>

char *asctime(const struct tm *tptr);
```

Description

This function returns an ASCII representation of the time represented by tptr. The string returned is always 26 characters and has this format:

```
Sun Jan 01 12:34:56 1993\n\0
```

The string pointed to is in a static buffer and will be overwritten with each call to asctime. The data should be copied if it needs to be preserved.

The layout of the `struct tm` structure is like this:

```
struct tm {
int tm_sec; /* seconds after the minute [0-60] */
int tm_min; /* minutes after the hour [0-59] */
int tm_hour; /* hours since 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 */
long tm_gmtoff; /* offset from GMT in seconds */
```

```
char * tm_zone; /* timezone abbreviation */
};
```

Return Value

A pointer to the string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
time_t now;
time(&now);
printf("The current time is %s", asctime(localtime(&now)));
```

asin

Syntax

```
#include <math.h>

double asin(double x);
```

Description

This function returns the angle in the range `[-Pi/2..Pi/2]` whose sine is x.

Return Value

The inverse sine, in radians, of x. If the absolute value of x is greater than 1, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

asinh

Syntax

```
#include <math.h>

double asinh(double x);
```

Description

This function returns the inverse hyperbolic sine of the argument x.

Return Value

The inverse hyperbolic sine of x. If the argument x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`. If x is a positive or negative `Inf`, the return value is equal to the value of x, and `errno` is left unchanged.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

assert

Syntax

```
#define NDEBUG
#include <assert.h>

assert(expression);
assertval(expression);
```

Description

These macros are used to assist in debugging. The source code includes references to `assert` or `assertval`, passing them expressions that should be `true` (non-zero). When the expression yields `false` (zero), a diagnostic message is printed to the standard error stream, and the program aborts.

If you define the macro `NDEBUG` before including `assert.h`, then these `assert` and `assertval` expand to nothing to reduce code size after debugging is done.

Return Value

`assert` returns 1 if its argument is non-zero, else it aborts.

`assertval` returns the value of its expression argument, if non-zero, else it aborts.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1) 1003.2-1992; 1003.1-2001 (see note 2)

Notes:

1. `assert` is ANSI, `assertval` is not.
2. `assert` is Posix, `assertval` is not.

Example

```
/* Like 'strdup', but doesn't crash if the argument is NULL. */
char * safe_strdup(const char *s)
{

assert(s != 0);
return strdup(s);
}
```

atan

Syntax

```
#include <math.h>

double atan(double x);
```

Description

This function computes the angle, in the range `[-Pi/2..Pi/2]` radians, whose tangent is x.

Return Value

The arc tangent, in radians, of x. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`. If x is a positive or negative `Inf`, the return value is equal to positive or negative `Pi/2`, respectively, and `errno` is left unchanged.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

atan2

Syntax

```
#include <math.h>

double atan2(double y, double x);
```

Description

This function computes the angle, in the range `[-Pi..Pi]` radians, whose tangent is `y/x`. In other words, it computes the angle, in radians, of the vector (x,y) with respect to the `+x` axis, reckoning the counterclockwise direction as positive, and returning the value in the range `[-Pi, Pi]`.

Return Value

The arc tangent, in radians, of `y/x`. `Pi` is returned if x is negative and y is a negative zero, `-0.0`. `-Pi` is returned, if x is negative, and y is a positive zero, `+0.0`.

If either x or y is infinite, `atan2` returns, respectively, `Pi` with the sign of y or zero, and `errno` is left unchanged. However, if *both* arguments are infinite, the return value is `NaN` and `errno` is set to `EDOM`.

A `NaN` is returned, and `errno` is set to `EDOM`, if either x and y are both zero, or if either one of the arguments is a `NaN`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

atanh

Syntax

```
#include <math.h>

double atanh(double x);
```

Description

This function computes the inverse hyperbolic tangent of x.

Return Value

The inverse hyperbolic tangent of x. If the the value of x is plus or minus 1, the return value is an `Inf` with the same sign as the argument x, and `errno` is set to `ERANGE`. If the absolute value of x is greater than 1, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

atexit

Syntax

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Description

This function places the specified function func on a list of functions to be called when `exit` is called. These functions are called as if a last-in-first-out queue is used, that is, the last function registered with `atexit` will be the first function called by `exit`.

At least 32 functions can be registered this way.

Return Value

Zero on success, non-zero on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
void exit_func()
{
```

```
remove("file.tmp");
}

...
atexit(exit_func);
...
```

atof

Syntax

```
#include <stdlib.h>

double atof(const char *string);
```

Description

Convert as much of the string as possible to an equivalent double precision real number.

This function is almost like `strtod(string, NULL)` (See strtod).

Return Value

The equivalent value, or zero if the string does not represent a number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
main(int argc, char **argv)
{

double d = atof(argv[1]);
...
```

atoi

Syntax

```
#include <stdlib.h>

int atoi(const char *string);
```

Description

Convert as much of the string as possible to an equivalent integer value.

This function is almost like `(int)strtol(string, NULL, 10)` (See strtol).

Return Value

The equivalent value, or zero if the string does not represent a number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
main(int argc, char **argv)
{

int i = atoi(argv[1]);
...
```

atol
Syntax
```
#include <stdlib.h>

long atol(const char *string);
```
Description
Convert as much of the string as possible to an equivalent long integer value.

This function is almost like `strtol(string, NULL, 10)` (See strtol).

Return Value
The equivalent value, or zero if the string does not represent a number.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example
```
main(int argc, char **argv)
{

long l = atol(argv[1]);
...
```
_atold
Syntax
```
#include <stdlib.h>

long double _atold(const char *string);
```
Description
Convert as much of the string as possible to an equivalent long double precision real number.

This function is almost like `_strtold(string, NULL)` (See _strtold).

Return Value
The equivalent value, or zero if the string does not represent a number.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
main(int argc, char **argv)
{

long double d = _atold(argv[1]);
...
```
atoll
Syntax
```
#include <stdlib.h>

long long int atoll(const char *string);
```
Description

Convert as much of the string as possible to an equivalent long long integer value.

This function is almost like `strtoll(string, NULL, 10)` (See strtoll).

Return Value

The equivalent value, or zero if the string does not represent a number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
main(int argc, char **argv)
{

long long int l = atoll(argv[1]);
...
}
```

basename

Syntax

```
#include <unistd.h>

char * basename (const char *fname);
```

Description

This function returns the basename of the file, which is the last part of its full name given by fname, with the drive letter and leading directories stripped off. For example, the basename of `c:/foo/bar/file.ext` is `file.ext`, and the basename of `a:foo` is `foo`. Trailing slashes and backslashes are significant: the basename of `c:/foo/bar/` is an empty string after the rightmost slash.

This function treats both forward- and backslashes like directory separators, so it can handle file names with mixed styles of slashes.

Return Value

A pointer into the original file name where the basename starts. Note that this is **not** a new buffer allocated with `malloc`. If fname is a NULL pointer, the function will return a NULL pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (strcmp (basename (file_name), "gcc.exe") == 0)
printf ("The file %s is the GNU C/C++ compiler\n", file_name);
```

bcmp

Syntax

```
#include <string.h>

int bcmp(const void *ptr1, const void *ptr2, int length);
```

Description

Compare memory pointed to by ptr1 and ptr2 for at most length bytes.

Return Value

The number of bytes remaining when the first mismatch occurred, or zero if all bytes were equal.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
void f(char *s1, char *s2)
{

int l = bcmp(s1, s2, strlen(s1));
printf("Difference: %s, %s\n", s1+strlen(s1)-l, s2+strlen(s1)-l);
}
```

bcopy
Syntax
```
#include <string.h>

void bcopy(const void *source, void *dest, int length);
```

Description
Copy length bytes from source to dest. Overlapping regions are handled properly, although this behavior is not portable.

Return Value
No value is returned.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
struct s a, b;
bcopy(a, b, sizeof(struct s));
```

bdos
Syntax
```
#include <dos.h>

int bdos(int func, unsigned dx, unsigned al);
```

Description
Calls function func of the software interrupt 0x21, passing it al as the subfunction and (the lower 16 bit of) dx in the `DX` register. This function will only work for a subset of DOS functions which require no arguments at all, or take non-pointer arguments in the `AL` and `DX` registers only. For functions which require a pointer in the `DX` register, use `bdosptr` (See bdosptr).

Return Value
Whatever the called function returns in the AX register.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
/* read a character */
int ch = bdos(1, 0, 0) & 0xff;
```

bdosptr

Syntax

```
#include <dos.h>

int bdosptr(int func, void *ptr, unsigned al);
```

Description

Calls function func of the software interrupt 0x21, passing it al as the subfunction and a pointer to a copy of the buffer contents whose address is in ptr through the `DX` register. This function will only work for a subset of DOS which require an argument in the `AL` register and a pointer in `DX` register. For functions which require non-pointer arguments in the `DX` register, use `bdos` (See bdos). To make the contents of ptr available to DOS, `bdosptr` copies it to the transfer buffer located in the low (below 1 Meg mark) memory.

Currently, some of the functions which take a pointer to a buffer in `DX` are *NOT* supported (notably, most of the FCB-based functions). See int86, for the list of supported functions.

Return Value

Whatever the called function returns in the AX register.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* print a string */
bdosptr(9, "Hello, there$", 0);
```

_bios_disk

Syntax

```
#include <bios.h>

unsigned _bios_disk(unsigned cmd, struct diskinfo_t *di)
```

Description

This function interfaces with the BIOS disk sevice (interrupt 0x13). The parameter cmd select the corresponding disk service and the structure di holds the disk parameters.

```
struct diskinfo_t {
unsigned drive; /* Drive number. */
unsigned head; /* Head number. */
unsigned track; /* Track number. */
unsigned sector; /* Sector number. (1-63) */
unsigned nsectors; /* Number of sectors to read/write/verify. */
void *buffer; /* Buffer for reading/writing/verifying. */
}
```

The following services are available based on value of cmd:

`_DISK_RESET`
: Forces the disk controller to do a hard reset, preparing for floppy-disk I/O. This is useful after an error occurs in another operation, such as a read. If this service is specified, the di argument is ignored. Status is returned in the 8 high-order bits (AH) of the return value. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

`_DISK_STATUS`
: Obtains the status of the last disk operation. If this service is specified, the <diskinfo> argument is ignored. Status is returned in the 8 low-order bits (AL) of the return value. If there is an error, the low-order byte (AL) will contain a set of status flags, as defined below under Return Value.

`_DISK_READ`
: Reads one or more disk sectors into memory. This service uses all fields of the structure pointed to by diskinfo. If no error occurs, the function returns 0 in the high-order byte and the number of sectors read in the low-order byte. If there is an error, the high-order byte (AH) will contain a set of status flags, as defined

below under Return Value.

_DISK_WRITE
: Writes data from memory to one or more disk sectors. This service uses all fields of the structure pointed to by <diskinfo>. If no error occurs, the function returns 0 in the high-order byte (AH) and the number of sectors written in the low-order byte (AL). If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_FORMAT
: Formats the track specified by diskinfo. The head and track fields indicate the track to format. Only one track can be formatted in a single call. The buffer field points to a set of sector markers. The format of the markers depends on the type of disk drive (see a technical reference to the PC BIOS to determine the marker format). The high-order byte (AH) of the return value contains the status of the call; 0 equals success. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_VERIFY
: Checks the disk to be sure the specified sectors exist and can be read. It also runs a CRC (cyclic redundancy check) test. This service uses all fields (except buffer) of the structure pointed to by diskinfo. If no error occurs, the function returns 0 in the high-order byte (AH) and the number of sectors compared in the low-order byte (AL), as defined below under Return Value.

Return Value

Return value of AX register. The meaning of high-order byte (AH):

```
0x00 No error
0x01 Invalid request or a bad command
0x02 Address mark not found
0x03 Disk write protected
0x04 Sector not found
0x05 Reset failed
0x06 Floppy disk removed
0x07 Drive parameter activity failed
0x08 Direct Memory Access (DMA) overrun
0x09 DMA crossed 64K boundary
0x0A Bad sector flag detected
0x0B Bad track flag detected
0x0C Media type not found
0x0D Invalid number of sectors on format
0x0E Control data access mark detected
0x0F DMA arbitration level out of range
0x10 Data read (CRC or ECC) error
0x11 Corrected data read (ECC) error
0x20 Controller failure
0x40 Seek error
0x80 Disk timed out or failed to respond
0xAA Drive not ready
0xBB Undefined error
0xCC Write fault on drive
0xE0 Status error
0xFF Sense operation failed
```

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char record_buffer[512];
struct diskinfo_t di;

di.drive = 0x80;
di.head = 0;
di.track = 0;
di.sector = 1;
di.nsectors = 1;
di.buffer = &record_buffer;
if ( _bios_disk(_DISK_READ, &di) )
puts("Disk error.");
```

_bios_equiplist

Syntax

```
#include <bios.h>

unsigned _bios_equiplist(void)
```

Description

This function returns the equipment word from BIOS request 0x11. The bits correspond to the following values:

```
Bits Meaning
0 True (1) if disk drive(s) installed
1 True (1) if math coprocessor installed
2-3 System RAM in 16K blocks (16-64K)
4-5 Initial video mode:
00 = Reserved
01 = 40 x 25 color
10 = 80 x 25 color
11 = 80 x 25 monochrome
6-7 Number of floppy-disk drives installed
(00 = 1, 01 = 2, etc.)
8 False (0) if and only if a Direct Memory Access (DMA)
chip is installed
9-11 Number of RS232 serial ports installed
12 True (1) if and only if a game adapter is installed
13 True (1) if and only if an internal modem is installed
14-15 Number of printers installed
```

Return Value

The equipment word.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if ( _bios_equiplist() & 0xc000 )
do_printing();
```

_bios_keybrd

Syntax

```
#include <bios.h>

unsigned _bios_keybrd(unsigned cmd);
```

Description

The `_bios_keybrd` function uses INT 0x16 to access the keyboard services. The cmd argument can be any of the following manifest constants:

`_KEYBRD_READ`
: read the next key pressed.

`_NKEYBRD_READ`
: read the next extended key pressed. Unlike `_KEYBRD_READ`, this command knows about keys introduced with the AT-style 101-key keyboards, such as **F11** and **F12**, and can distinguish between the editing keys on the numeric pad and the grey keys of the edit pad. On the other hand, some of the extended keys return two-byte sequences which typically begin with the `E0h` (224 decimal) prefix, so code that uses `_NKEYBRD_READ` should deal with this complexity.

`_KEYBRD_READY`
: check if a key is waiting in the keyboard buffer.

`_NKEYBRD_READY`
: check if an extended key is waiting in the keyboard buffer. Like `_KEYBRD_READY`, but recognizes extended

keys such as **F12**, which _KEYBRD_READY ignores.

_KEYBRD_SHIFTSTATUS
read keyboard shift state (the byte at the address 40h:17h in the BIOS data area):

```
7654 3210 Meaning
---- ---X Right SHIFT is pressed
---- --X- Left SHIFT is pressed
---- -X-- CTRL is pressed
---- X--- ALT is pressed
---X ---- Scroll Lock locked
--X- ---- Num Lock locked
-X-- ---- Caps Lock locked
X--- ---- Insert locked
```

_NKEYBRD_SHIFTSTATUS
read keyboard shift and extended shift state (the byte at the address 40h:17h in the BIOS data area combined with the extended shift flags from the bytes at addresses 40h:18h and 40h:96h):

```
FEDC BA98 7654 3210 Meaning
---- ---- ---- ---X Right SHIFT is pressed
---- ---- ---- --X- Left SHIFT is pressed
---- ---- ---- -X-- CTRL is pressed
---- ---- ---- X--- ALT is pressed
---- ---- ---X ---- Scroll Lock locked
---- ---- --X- ---- Num Lock locked
---- ---- -X-- ---- Caps Lock locked
---- ---- X--- ---- Insert locked

---- ---X ---- ---- Left CTRL is pressed
---- --X- ---- ---- Left ALT is pressed
---- -X-- ---- ---- Right CTRL is pressed
---- X--- ---- ---- Right ALT is pressed
---X ---- ---- ---- Scroll Lock is pressed
--X- ---- ---- ---- Num Lock is pressed
-X-- ---- ---- ---- Caps Lock is pressed
X--- ---- ---- ---- SysReq is pressed
```

Return Value

With the *_READ and *_SHIFTSTATUS arguments, the _bios_keybrd function returns the contents of the AX register after the BIOS call. For the *_READ arguments, this is the combination of scan code and ASCII code for alphanumeric keys, or a scan code and either zero or the E0h prefix for special keys.

With the *_READY arguments, _bios_keybrd returns 0 if no key is waiting in the BIOS keyboard buffer. If there is a key, _bios_keybrd returns the key waiting to be read (that is, the same value as the corresponding *_READ would return).

With the *_READ and *_READY arguments, the _bios_keybrd function returns -1 if **Ctrl+BREAK** has been pressed and is the next keystroke to be read.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while( !_bios_keybrd(_KEYBRD_READY) )
try_to_do_something();
```

_bios_memsize

Syntax

```
#include <bios.h>

unsigned _bios_memsize(void);
```

Description

This function returns the amount of system memory in 1K blocks (up to 640K).

Return Value

Size of memory (in K).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("This system has %d bytes of memory\n", _bios_memsize() * 1024);
```

_bios_printer

Syntax

```
#include <bios.h>

unsigned _bios_printer(unsigned cmd, unsigned printer, unsigned data);
```

Description

The _bios_printer routine uses INT 0x17 to perform printer output services for parallel printers. The printer argument specifies the affected printer, where 0 is LPT1, 1 is LPT2, and so on. The cmd argument can be any of the following manifest constants:

`_PRINTER_INIT`
`Reset and initialize the specified printer port`
`_PRINTER_STATUS`
Return the status of the specified printer port

`_PRINTER_WRITE`
Print the data argument to the specified printer port

Return Value

The _bios_printer function returns the value in the AX register after the BIOS interrupt. The high-order byte (AH) of the return value indicates the printer status after the operation, as defined below:

```
Bit Meaning if True

0 Printer timed out
1 Not used
2 Not used
3 I/O error
4 Printer selected
5 Out of paper
6 Acknowledge
7 Printer not busy
```

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (*c)
_bios_printer(_PRINTER_WRITE, *c++, 0);
```

_bios_serialcom

Syntax

```
#include <bios.h>

unsigned _bios_serialcom(unsigned cmd, unsingned serialport,
unsigned data);
```

Description

The _bios_serialcom routine uses INT 0x14 to provide serial communications services. The serialport argument is set to 0 for COM1, to 1 for COM2, and so on. The cmd argument can be set to one of the following manifest constants:

`_COM_INIT`
  Initialize com port (data is the settings)

`_COM_RECEIVE`
  Read a byte from port

`_COM_SEND`
  Write a byte to port

`_COM_STATUS`
  Get the port status

The data argument is ignored if cmd is set to _COM_RECEIVE or _COM_STATUS. The data argument for _COM_INIT is created by combining one or more of the following constants (with the OR operator):

```
_COM_CHR7 7 bits/character
_COM_CHR8 8 bits/character
_COM_STOP1 1 stop bit
_COM_STOP2 2 stop bits
_COM_NOPARITY no parity
_COM_EVENPARITY even parity
_COM_ODDPARITY odd parity
_COM_110 110 baud
_COM_150 150 baud
_COM_300 300 baud
_COM_600 600 baud
_COM_1200 1200 baud
_COM_2400 2400 baud
_COM_4800 4800 baud
_COM_9600 9600 baud
```

The default value of data is 1 stop bit, no parity, and 110 baud.

Return Value

The function returns a 16-bit integer whose high-order byte contains status bits. The meaning of the low-order byte varies, depending on the cmd value. The high-order bits are as follows:

```
Bit Meaning if Set

15 Timed out
14 Transmission-shift register empty
13 Transmission-hold register empty
12 Break detected
11 Framing error
10 Parity error
9 Overrun error
8 Data ready
```

When service is _COM_SEND, bit 15 is set if data cannot be sent.

When service is _COM_RECEIVE, the byte read is returned in the low-order bits if the call is successful. If an error occurs, any of the bits 9, 10, 11, or 15 is set.

When service is _COM_INIT or _COM_STATUS, the low-order bits are defined as follows:

```
Bit Meaning if Set

7 Receive-line signal detected
6 Ring indicator
5 Data-set-ready
4 Clear-to-send
3 Change in receive-line signal detected
2 Trailing-edge ring indicator
```

```
1 Change in data-set-ready status
0 Change in clear-to-send status
```

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* 9600 baud, no parity, one stop, 8 bits */
_bios_serialcom(_COM_INIT, 0,
_COM_9600|_COM_NOPARITY|_COM_STOP1|_COM_CHR8);
for(i=0; buf[i]; i++)
_bios_serialcom(_COM_SEND, 0, buf[i]);
```

_bios_timeofday

Syntax

```
#include <bios.h>

unsigned _bios_timeofday(unsigned cmd, unsigned long *timeval);
```

Description

The _bios_timeofday routine uses INT 0x1A to get or set the clock count (which is the number of 18.2 Hz ticks since midnight). The cmd argument can be either the _TIME_GETCLOCK or _TIME_SETCLOCK manifest constant.

Return Value

If the argument is _TIME_GETCLOCK, the routine returns a nonzero value if midnight was passed since last read, or zero if midnight was not passed. If the argument is _TIME_SETCLOCK, the return value is undefined.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned hour, min, sec, hsec;
unsigned long ticks;
...
ticks = (unsigned long)(hour * 65543.33) + (min * 1092.38) +
(sec * 18.21) + (hsec * 0.182);
_bios_timeofday(_TIME_SETCLOCK, &ticks);
```

bioscom

Syntax

```
#include <bios.h>

int bioscom(int cmd, char data, int port);
```

Description

This function accesses the BIOS interrupt 0x14 function, serial communication services. `port` should be the COM port (0=COM1, 1=COM2, etc).

The valid values of cmd are:

0 - initialize com port (data is the settings)
1 - write byte to port
2 - read byte from port (data is ignored)
3 - get port status

For initialization, the byte is made up of the following bits:

```
0000 0000
7654 3210 Meaning
```

```
---- --10 7 bits/character
---- --11 8 bits/character
---- -0-- 1 stop bit
---- -1-- 2 stop bits
---X 0--- no parity
---0 1--- odd parity
---1 1--- even parity
000- ---- 110 baud
001- ---- 150 baud
010- ---- 300 baud
011- ---- 600 baud
100- ---- 1200 baud
101- ---- 2400 baud
110- ---- 4800 baud
111- ---- 9600 baud
```

For writing a character out to the port, the return value's lower 8 bits contain the same byte as passed as the data argument.

For reading a character from the port, the value of data is ignored, and the lower 8 bits of the return value contain the byte read. Also, the "timeout" bit in the upper 8 bits is used as an error indicator in this case (0=success, 1=error). If it indicates an error, you should call the "get port status" variant to get the detailed error bits.

Return Value

The return value is a sequence of bits that indicate the port status and, for cmd=0 and 3, the modem status. For read/write operations, the lower eight bits are the character read.

```
1111 1100 0000 0000
5432 1098 7654 3210 Meaning

---- ---- ---- ---1 CTS change
---- ---- ---- --1- DSR change
---- ---- ---- -1-- ring change
---- ---- ---- 1--- carrier detect change
---- ---- ---1 ---- CTS present
---- ---- --1- ---- DSR present
---- ---- -1-- ---- ring present
---- ---- 1--- ---- carrier detect
---- ---1 ---- ---- data ready
---- --1- ---- ---- overrun error
---- -1-- ---- ---- parity error
---- 1--- ---- ---- framing error
---1 ---- ---- ---- break detected
--1- ---- ---- ---- transmit holding register empty
-1-- ---- ---- ---- transmit shift register empty
1--- ---- ---- ---- time out (=1 if error present for cmd=1,2)
```

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
bioscom(0, 0xe3, 0); /* 9600 baud, no parity, one stop, 8 bits */
for (i=0; buf[i]; i++)
bioscom(1, buf[i], 0);
```

biosdisk

Syntax

```
#include <bios.h>

int biosdisk(int cmd, int drive, int head, int track,
int sector, int nsects, void *buffer);
```

Description

This function interfaces with the BIOS disk service (interrupt 0x13). Please refer to a BIOS reference manual for detailed information about the parameters of this call. The function assumes a sector size of 512 bytes.

The following functions of Int 13h are currently supported:

0 - reset disk subsystem
1 - get status of last operation (see See _bios_disk, for possible values)
2 - read one or more sectors
3 - write one or more sectors
5 - format a track
6 - format a cylinder and set bad sector flag
7 - format drive from specified cylinder
8 - get drive parameters
9 - initialize drive parameters
10 - read long sectors
11 - write long sectors
12 - seek to cylinder
13 - alternate fixed disk reset
14 - read sector buffer
15 - write sector buffer
16 - test for drive ready
17 - recalibrate drive
18 - controller RAM diagnostic
19 - controller drive diagnostic
20 - controller internal diagnostic
21 - get DASD type
22 - read disk change line status
23 - set DASD type (pass DASD code in nsects)
24 - set media type for format

The first request with more sectors than will fit in the transfer buffer will cause a DOS buffer to be allocated. This buffer is automatically freed when your application exits. Requests for more sectors than 18 sectors (9K) will fail.

Function 8 returns values in buffer as follows:

byte 0 = sectors per track (bits 0..5), top two bits of cylinder (in bits 6..7)
byte 1 = cylinders (bits 0..7)
byte 2 = number of drives
byte 3 = number of heads

Return Value

The value of AH returned by the BIOS. See See _bios_disk, for a detailed list of possible status and error codes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buffer[512];
if (biosdisk(2, 0x80, 0, 0, 0, 1, buffer))
error("disk");
```

biosequip

Syntax

```
#include <bios.h>

int biosequip(void);
```

Description

This function returns the equipment word from BIOS request 0x11. The bits correspond to the following values:

```
1111 1100 0000 0000
5432 1098 7654 3210 Meaning
```

```
---- ---- ---- ---X 1 = disk drive(s) installed
---- ---- ---- --X- 1 = math coprocessor installed
---- ---- ---- XX-- System memory: 00=16k, 01=32k, 10=48k,
11=64k (non PS/2)
---- ---- ---- -X-- 1 = pointing device installed (PS/2)
---- ---- ---- X--- not used on PS/2
---- ---- --XX ---- initial video mode: 01=CO40 10=CO80 11=MONO
---- ---- XX-- ---- disk drives 00=1 01=2 10=3 11=4 (zero if bit 1=0)
---- ---X ---- ---- 1 = no DMA available
---- XXX- ---- ---- number of serial ports installed (000=0 001=1 etc)
---X ---- ---- ---- 1 = game port adapter installed
--X- ---- ---- ---- 1 = internal modem installed (PS/2)
--X- ---- ---- ---- 1 = serial printer attached (non PS/2)
XX-- ---- ---- ---- number of printers installed (00=0 01=1 10=2 11=3)
```

Return Value

The equipment word.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (biosequip() & 0xc000)
do_printing();
```

bioskey

Syntax

```
#include <bios.h>

int bioskey(int command)
```

Description

This function issues the BIOS keyboard interrupt 16h with command in the AH register, and returns the results of that call. The argument command can accept the following values:

command = 00h

Returns the next key pressed. The value returned is a combination of the key's scan code in the high 8 bits and its ASCII code in the low 8 bits. For non-alphanumeric keys, such as the arrow keys, the low 8 bits are zeroed.

command = 01h

Checks the keyboard, returns zero if no key pressed, else the key. Does not dequeue the key from the keyboard buffer. The value returned when a key was pressed is a combination of the key's scan code in the high 8 bits and either its ASCII code or zero in the low 8 bits.

If the **Ctrl-BREAK** key was pressed, returns -1.

command = 02h

Returns the keyboard shift state:

```
7654 3210 Meaning

---- ---X Right shift key down
---- --X- Left shift key down
---- -X-- Ctrl key down
---- X--- Alt key down
---X ---- Scroll lock on
--X- ---- Num lock on
-X-- ---- Caps lock on
X--- ---- Insert on
```

command = 10h

Returns the next extended key pressed. This works like the case of command = 0, but it recognizes additional keys from the AT-style extended 101-key keyboard, like the second **Alt** key and **F12**. If a key was pressed, returns the scan code and ASCII code packed in same way as for command = 0, except that the extended keys have the `E0h` prefix in the low 8 bits.

Almost every PC nowadays has an extended 101-key keyboard.

command = 11h
Like the case of command = 1, but recognizes the additional keys of the extended keyboard.

command = 12h
Returns the two status bytes of the enhanced keyboard, packed in the low 16 bits of the return value. The individual bits of the return value are defined in the following table:

```
FEDC BA98 7654 3210 Meaning
---- ---- ---- ---X Right SHIFT is pressed
---- ---- ---- --X- Left SHIFT is pressed
---- ---- ---- -X-- CTRL is pressed
---- ---- ---- X--- ALT is pressed
---- ---- ---X ---- Scroll Lock locked
---- ---- --X- ---- Num Lock locked
---- ---- -X-- ---- Caps Lock locked
---- ---- X--- ---- Insert locked

---- ---X ---- ---- Left CTRL is pressed
---- --X- ---- ---- Left ALT is pressed
---- -X-- ---- ---- Right CTRL is pressed
---- X--- ---- ---- Right ALT is pressed
---X ---- ---- ---- Scroll Lock is pressed
--X- ---- ---- ---- Num Lock is pressed
-X-- ---- ---- ---- Caps Lock is pressed
X--- ---- ---- ---- SysReq is pressed
```

Return Value

Depends on command.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (!bioskey(1))
do_stuff();
```

biosmemory

Syntax

```
#include <bios.h>

unsigned biosmemory(void);
```

Description

This function returns the amount of system memory in 1k blocks.

Note that this function doesn’t know about extended memory above the 640K mark, so it will report 640K at most. This is a limitation of the BIOS.

Return Value

Bytes of memory / 1024.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("This system has %d bytes of memory\n", biosmemory()*1024);
```

biosprint

Syntax

```
#include <bios.h>

int biosprint(int cmd, int byte, int port)
```

Description

command = 0
`byte` is sent to parallel port port.

command = 1
Parallel port port is reset and initialized.

command = 2
The status of parallel port port is returned.

```
7654 3210 Meaning

---- ---X Timeout
---- -XX- Unused
---- X--- I/O Error
---X ---- Selected
--X- ---- Out of paper
-X-- ---- Acknowledged
X--- ---- Idle
```

Return Value

The printer status.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (*c)
biosprint(0, *c++, 0);
```

biostime

Syntax

```
#include <bios.h>

long biostime(int cmd, long newtime);
```

Description

This function reads (cmd=0) or sets (cmd=1) the internal tick counter, which is the number of 18.2 Hz ticks since midnight.

Return Value

When reading, the number of ticks since midnight.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
long ticks = biostime(0, 0);
```

blinkvideo

Syntax

```
#include <conio.h>

void blinkvideo(void);
```

Description

Bit 7 (`MSB`) of the character attribute byte has two possible effects on EGA and VGA displays: it can either make the character blink or change the background color to bright (thus allowing for 16 background colors as opposed to the usual 8). This function sets that bit to display blinking characters. After a call to this function, every character written to the screen with bit 7 of the attribute byte set, will blink. The companion function `intensevideo` (See intensevideo) has the opposite effect.

Note that there is no BIOS function to get the current status of this bit, but bit 5 of the byte at `0040h:0065h` in the BIOS area indicates the current state: if it's 1 (the default), blinking characters will be displayed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

brk

Syntax

```
#include <unistd.h>

int brk(void *ptr);
```

Description

This function changes the *break* for the program. This is the first address that, if referenced, will cause a fault to occur. The program asks for more memory by specifying larger values for ptr. Normally, this is done transparently through the `malloc` function.

Return Value

Zero if the break was changed, -1 if not. `errno` is set to the error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (brk(old_brk+1000))
printf("no memory\n");
```

bsearch

Syntax

```
#include <stdlib.h>

void *bsearch (const void *key, const void *base, size_t num,
size_t size, int (*ptf)(const void *ckey, const void *celem));
```

Description

Given an array of values, perform a binary search on the values looking for value that "matches" the given key. A match is determined by calling the provided function ptf and passing it the key as ckey and a pointer to one of the elements of the array as celem. This function must return a negative number if the key is closer than the element to the beginning of the array, positive if it is closer to the end, and zero if the element matches the key.

The array begins at address base and contains num elements, each of size size.

Return Value

Returns a pointer to the element that matches the key, else NULL.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
typedef struct {
int a, b;
} q;

int compare(void *key, void *elem)
{

return *(int *)key - ((q *)elem)->a;
}


q qlist[100];

...
q *match = bsearch(4, qlist, 100, sizeof(q), compare);
printf("4->%d=n", match->b);
...
```

bzero

Syntax

```
#include <string.h>

void bzero(void *pointer, int length);
```

Description

The data at pointer is filled with length zeros.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char foo[100];
bzero(foo,100);
```

calloc

Syntax

```
#include <stdlib.h>

void *calloc(size_t num_elements, size_t size);
```

Description

This function allocates enough memory for num_elements objects of size size. The memory returned is initialized to all zeros. The pointer returned should later be passed to free (See free) so that the memory can be returned to the heap.

You may use cfree (See cfree) to free the pointer also; it just calls free.

Return Value

A pointer to the memory, or `NULL` if no more memory is available.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
Complex *x = calloc(12, sizeof(Complex));
cfree(x);
```

cbrt

Syntax

```
#include <math.h>

double cbrt(double x);
```

Description

This function computes the cube root of x. It is faster and more accurate to call `cbrt(x)` than to call `pow(x, 1./3.)`.

Return Value

The cube root of x. If the value of x is `NaN`, the return value is `NaN` and `errno` is set to `EDOM`. Infinite arguments are returned unchanged, without setting `errno`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ceil

Syntax

```
#include <math.h>

double ceil(double x);
```

Description

This function computes the smallest integer greater than or equal to x.

Return Value

The smallest integer value greater than or equal to x.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

cfgetispeed

Syntax

```
#include <termios.h>

speed_t cfgetispeed (const struct termios *termiosp);
```

Description

This function gets the input line speed stored in the structure termiosp. See Termios functions, for more details about this structure and the baudrate values it supports.

Note that the termios emulation handles console only, and that the input baudrate value is ignored by this implementation.

Return Value

The input line speed on success, (speed_t) -1 for error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

cfgetospeed

Syntax

```
#include <termios.h>

speed_t cfgetospeed (const struct termios *termiosp);
```

Description

This function gets the output line speed stored in the structure termiosp. See Termios functions, for more details about this structure and the baudrate values it supports.

Note that the termios emulation handles console only, and that the baudrate value has no effect in this implementation.

Return Value

The output line speed on success, (speed_t) -1 for error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

cfmakeraw

Syntax

```
#include <termios.h>

void cfmakeraw (struct termios *termiosp);
```

Description

This function sets the structure specified by termiosp for raw mode. It is provided for compatibility only. Note that the termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

cfree

Syntax

```
#include <stdlib.h>

void cfree(void *pointer);
```

Description

This function returns the memory allocated by calloc (See calloc) to the heap.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
Complex *x = calloc(12, sizeof(Complex));
cfree(x);
```

cfsetispeed

Syntax

```
#include <termios.h>

int cfsetispeed (struct termios *termiosp, speed_t speed);
```

Description

This function sets the input line speed stored in the structure termiosp to speed. See Termios functions, for more details about this structure and the baudrate values it supports.

Note that the termios emulation handles console only, and that the baudrate values have no effect in this implementation.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

cfsetospeed

Syntax

```
#include <termios.h>

int cfsetospeed (struct termios *termiosp, speed_t speed);
```

Description

This function sets the output line speed stored in the structure termiosp to speed. See Termios functions, for more details about this structure and the baudrate values it supports.

Note that the termios emulation handles console only, and that the baudrate values have no effect in this implementation.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

cfsetspeed

Syntax

```
#include <termios.h>

int cfsetspeed (struct termios *termiosp, speed_t speed);
```

Description

This function sets the input and output line speed stored in the structure termiosp to speed. It is provided for compatibility only. Note that the termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

cgets

Syntax

```
#include <conio.h>

char *cgets(char *_str);
```

Description

Get a string from the console. This will take advantage of any command-line editing TSRs. To use, you must pre-fill the first character of the buffer. The first character is the size of the buffer. On return, the second character is the number of characters read. The third character is the first character read.

Return Value

A pointer to the first character read.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

chdir

Syntax

```
#include <unistd.h>

int chdir(const char *new_directory);
```

Description

This function changes the current directory to new_directory. If a drive letter is specified, the current directory for that drive is changed and the current disk is set to that drive, else the current directory for the current drive is changed.

Return Value

Zero if the new directory exists, else nonzero and `errno` set if error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
if (chdir("/tmp"))
perror("/tmp");
```

_check_v2_prog

Syntax

```
#include <sys/system.h>

const _v2_prog_type *_check_v2_prog(const char *program, int fd);
```

Description

This function checks a given program for various known types of executables and/or other things. This function povides two differnt entry points. One is to call the function with a not `NULL` pointer as `program` (in this case `fd` is ignored), then the file named by `program` is opened and closed by `_check_v2_prog`.

When you pass `NULL` as `program`, then you have to pass a valid file handle in `fd` and `_check_v2_prog` uses that handle and does also not close the file on return.

Return Value

`_v2_prog_type` is defined in `sys/system.h` like the following:

```
typedef struct {
char magic[16];
int struct_length;
char go32[16];
unsigned char buffer[0];
} _v1_stubinfo;


typedef struct {
union {
unsigned version:8; /* The version of DJGPP created that COFF exe */
struct {
unsigned minor:4; /* The minor version of DJGPP */
unsigned major:4; /* The major version of DJGPP */
} v;
} version;

unsigned object_format:4; /* What an object format */
# define _V2_OBJECT_FORMAT_UNKNOWN 0x00
# define _V2_OBJECT_FORMAT_COFF 0x01
# define _V2_OBJECT_FORMAT_PE_COFF 0x02

unsigned exec_format:4; /* What an executable format */
# define _V2_EXEC_FORMAT_UNKNOWN 0x00
# define _V2_EXEC_FORMAT_COFF 0x01
# define _V2_EXEC_FORMAT_STUBCOFF 0x02
# define _V2_EXEC_FORMAT_EXE 0x03
# define _V2_EXEC_FORMAT_UNIXSCRIPT 0x04

unsigned valid:1; /* Only when nonzero all the information is valid */

unsigned has_stubinfo:1; /* When nonzero the stubinfo info is valid */

unsigned unused:14;

_v1_stubinfo *stubinfo;
} _v2_prog_type;
```

The macros shown above can be used to test the different members of that structure for known values.

Warning: Do not modify any of the data in this structure.

After calling `_check_v2_prog` you should check at first the member `valid`. Only if this is nonzero you can be sure that all the other information in the struct is valid.

The same is for the `stubinfo` member of the above struct, it is valid only, when `has_stubinfo` is nonzero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

To use the information returned in the struct you can use code like the following:

```
#include <stdio.h>
#include <sys/system.h>

int main()
{

const _v2_prog_type *type;
/* Since we pass a valid name, we can use -1 as the second argument */
type = _check_v2_prog ("foo", -1);

/* There was something wrong */
if (!type->valid)
{
fprintf(stderr, "Could not check the file ’foo’. Giving up.\\n");
return 1;
}

/* Currently only the COFF format is valid to be a V2 executable */
if (type->object_format != _V2_OBJECT_FORMAT_COFF)
{
fprintf(stderr, "File ’foo’ is not in COFF format\\n");
return 2;
}

/* The major version is not 2 */
if (type->version.v.major != 2)
{
fprintf(stderr, "File ’foo’ is not from DJGPP 2.xx\\n");
return 3;
}

fprintf(stdout, "File ’foo’ is a valid DJGPP 2.xx executable\\n");

if (type->exec_format == _V2_EXEC_FORMAT_STUBCOFF)
{
fprintf(stdout, "File ’foo’ has a stub loader prepended\\n");
}

return 0;
}
```

chmod

Syntax

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

Description

This function changes the mode (writable or write-only) of the specified file. The value of mode can be a combination of one or more of the following:

`S_IRUSR`
Make the file readable for the owner.

`S_IWUSR`
Make the file writable for the owner.

`S_IRGRP`
Make the file readable for the group.

`S_IWGRP`

Make the file writeable for the group.

`S_IROTH`
Make the file readable for the world.

`S_IWOTH`
Make the file writeable for the world.

Some `S_I*` constants are ignored for regular files:

- `S_I*GRP` and `S_I*OTH` are ignored, because DOS/Windows has no concept of ownership, so all files are considered to belong to the user;
- `S_IR*` are ignored, because files are always readable on DOS/Windows.

This function can be hooked by File System Extensions (See File System Extensions).

Return Value

Zero if the file exists and the mode was changed, else nonzero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
chmod("/tmp/dj.dat", S_IWUSR|S_IRUSR);
```

_chmod

Syntax

```
#include <io.h>
int _chmod(const char *filename, int func, mode_t mode);
```

Description

This is a direct connection to the MS-DOS chmod function call, int 0x21, %ax = 0x4300/0x4301. If func is 0, then DOS is called with AX = 0x4300, which returns an attribute byte of a file. If func is 1, then the attributes of a file are set as specified in mode. Note that the directory and volume attribute bits must always be 0 when `_chmod()` is called with func = 1, or else the call will fail. The third argument is optional when getting attributes. The attribute bits are defined as follows:

```
Bits Meaning
76543210
.......1 Read-only
......1. Hidden
.....1.. System
....1... Volume Label
...1.... Directory
..1..... Archive
xx...... Reserved (used by some network redirectors)
```

On platforms where the LFN API (See _use_lfn, LFN) is available, `_chmod` calls function 0x7143 of Interrupt 21h, to support long file names.

Return Value

If the file exists, `_chmod()` returns its attribute byte in case it succeded, or -1 in case of failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

chown

Syntax

```
#include <unistd.h>

int chown(const char *file, int owner, int group);
```

Description

This function changes the ownership of the open file specified by file to the user ID owner and group ID group.

This function does nothing under MS-DOS. This function can be hooked by File System Extensions (See File System Extensions).

Return Value

This function always returns zero if the file exists, else it returns -1 and sets `errno` to `ENOENT`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

chsize

Syntax

```
#include <io.h>

int chsize(int handle, long size);
```

Description

Just calls ftruncate (See ftruncate).

Return Value

Zero on success, -1 on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

cleanup_client

Syntax

```
#include <debug/dbgcom.h>

void cleanup_client (void);
```

Description

This function is typically called when the debugged process exits or is aborted. It restores segment descriptors, closes file handles that were left open by the debuggee, frees protected-mode and conventional memory and any segment descriptors allocated by the debuggee, and restores the debugger's original signal handlers.

_clear87

Syntax

```
#include <float.h>

unsigned int _clear87(void);
```

Description

Clears the floating point processor's exception flags.

Return Value

The previous status word.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__clear_fd_flags

Syntax

```
#include <libc/fd_props.h>

void __clear_fd_flags(int fd, unsigned long flags);
```

Description

This internal function clears the combination of flags flags from the flags associated with the file descriptor fd. The flags are some properties that may be associated with a file descriptor (See __set_fd_properties).

The caller should first check that fd has properties associated with it, by calling `__has_fd_properties` (See __has_fd_properties).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__clear_fd_properties

Syntax

```
#include <libc/fd_props.h>

int __clear_fd_properties(int fd);
```

Description

This internal function is called when the file descriptor fd is no longer valid. The usage count of the associated `fd_properties` struct is decremented. And if it becomes zero, this function performs cleanup and releases the memory used by the `fd_properties` struct.

For more information, see `__set_fd_properties` (See __set_fd_properties) and `__dup_fd_properties` (See __dup_fd_properties).

Return Value

Always returns 0 for success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

clearerr

Syntax

```
#include <stdio.h>

void clearerr(FILE *stream);
```

Description

This function clears the EOF and error indicators for the file stream.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
clearerr(stdout);
```

clock

Syntax

```
#include <time.h>

clock_t clock(void);
```

Description

This function returns the number of clock ticks since an arbitrary time, actually, since the first call to `clock`, which itself returns zero. The number of tics per second is CLOCKS_PER_SEC.

Return Value

The number of tics.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
printf("%d seconds have elapsed\n", clock()/CLOCKS_PER_SEC);
```

close

Syntax

```
#include <unistd.h>

int close(int fd);
```

Description

The open file associated with fd is closed.

Return Value

Zero if the file was closed, nonzero if fd was invalid or already closed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int fd = open("data", O_RDONLY);
close(fd);
```

_close

Syntax

```
#include <io.h>

int _close(int fd);
```

Description

This is a direct connection to the MS-DOS close function call, int 0x21, %ah = 0x3e. This function can be hooked by the See File System Extensions. If you don't want this, you should use See _dos_close.

Return Value

Zero if the file was closed, else nonzero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

closedir

Syntax

```
#include <dirent.h>

int closedir(DIR *dir);
```

Description

This function closes a directory opened by opendir (See opendir).

Return Value

Zero on success, nonzero if dir is invalid.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

clreol

Syntax

```
#include <conio.h>

void clreol(void);
```

Description

Clear to end of line.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

clrscr

Syntax

```
#include <conio.h>

void clrscr(void);
```

Description

Clear the entire screen.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

confstr

Syntax

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

Description

This function stores various system-dependent configuration values in buf. name is one of the following:

`_CS_PATH`
: Returns a path to the standard POSIX utilities.

`_CS_POSIX_V6_ILP32_OFF32_CFLAGS`
: Returns the compile-time flags required to build an application using 32-bit `int`, `long`, pointer, and `off_t` types.

`_CS_POSIX_V6_ILP32_OFF32_LDFLAGS`
: Returns the link-time flags required to build an application using 32-bit `int`, `long`, pointer, and `off_t` types.

`_CS_POSIX_V6_ILP32_OFF32_LIBS`
: Returns the set of libraries required to build an application using 32-bit `int`, `long`, pointer, and `off_t` types.

If len is not zero and name has a defined value, that value is copied into buf and null terminated. If the length of the string to be copied plus the null terminator is greater than len bytes, the string is truncated to len-1 bytes and the result is null terminated.

If len is zero, nothing is copied into buf and the size of the buffer required to store the string is returned.

Return Value

If name has a defined value, the minimum size of the buffer required to hold the string including the terminating null is returned. If this value is greater than len, then buf is truncated.

If name is valid but does not have a defined value, zero is returned.

If name is invalid, zero is returned and `errno` is set to `EINVAL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
char *path;
size_t path_len;

path_len = confstr (_CS_PATH, NULL, 0);
path = malloc(path_len);
confstr(_CS_PATH, path, path_len);
```

_conio_gettext

Syntax

```
#include <conio.h>

int _conio_gettext(int _left, int _top, int _right, int _bottom, void *_destin);
```

Description

Retrieve a block of screen characters into a buffer.

Return Value

1

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_conio_kbhit

Syntax

```
#include <conio.h>

int _conio_kbhit(void);
```

Description

Determines whether or not a character is waiting at the keyboard. If there is an ungetch'd character, this function returns true. Note that if you include `conio.h`, the `kbhit` (See kbhit) function is redefined to be this function instead.

Return Value

Nonzero if a key is waiting, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

_control87

Syntax

```
#include <float.h>

unsigned int _control87(unsigned int newcw, unsigned int mask);
```

Description

This function sets and retrieves the FPU's control word.

The control word is a special 16-bit register maintained by the math coprocessor. By setting and clearing bit fields in the control word, you can exercise control of certain aspects of coprocessor operation. The individual bits of the x87 control word are defined by macros in float.h, and shown in this table:

```
---- ---- --XX XXXX = MCW_EM - exception masks (1=handle exception
internally, 0=fault)
---- ---- ---- ---X = EM_INVALID - invalid operation
---- ---- ---- --X- = EM_DENORMAL - denormal operand
---- ---- ---- -X-- = EM_ZERODIVIDE - divide by zero
---- ---- ---- X--- = EM_OVERFLOW - overflow
---- ---- ---X ---- = EM_UNDERFLOW - underflow
---- ---- --X- ---- = EM_INEXACT - rounding was required
---- --XX ---- ---- = MCW_PC - precision control
---- --00 ---- ---- = PC_24 - single precision
---- --10 ---- ---- = PC_53 - double precision
---- --11 ---- ---- = PC_64 - extended precision
---- XX-- ---- ---- = MCW_RC - rounding control
---- 00-- ---- ---- = RC_NEAR - round to nearest
```

```
---- 01-- ---- ---- = RC_DOWN - round towards -Inf
---- 10-- ---- ---- = RC_UP - round towards +Inf
---- 11-- ---- ---- = RC_CHOP - round towards zero
---X ---- ---- ---- = MCW_IC - infinity control (obsolete,
always affine)
---0 ---- ---- ---- = IC_AFFINE - -Inf < +Inf
---1 ---- ---- ---- = IC_PROJECTIVE - -Inf == +Inf
```

`_control87` uses the value of newcw and mask variables together to determine which bits of the FPU's control word should be set, and to what values. For each bit in mask that is set (equals to 1), the corresponding bit in newcw specifies the new value of the same bit in the FPU's control word, which `_control87` should set. Bits which correspond to reset (zero) bits in mask are not changed in the FPU's control word. Thus, using a zero value for mask retrieves the current value of the control word without changing it.

The exception bits `MCW_EM` (the low-order 6 bits) of the control word define the exception *mask*. That is, if a certain bit is set, the corresponding exception will be masked, i.e., it will not generate an FP exception (which normally causes signal `SIGFPE` to be delivered). A masked exception will be handled internally by the coprocessor. In general, that means that it will generate special results, such as NaN, Not-a-Number (e.g., when you attempt to compute a square root of a negative number), denormalized result (in case of underflow), or infinity (e.g., in the case of division by zero, or when the result overflows).

By default, DJGPP startup code masks all FP exceptions.

The precision-control field `MCW_PC` (bits 8 and 9) controls the internal precision of the coprocessor by selecting the number of precision bits in the mantissa of the FP numbers. The values `PC_24`, `PC_53`, and `PC_64` set the precision to 24, 53, and 64-bit mantissa, respectively. This feature of the coprocessor is for compatibility with the IEEE 745 standard and only affect the `FADD`, `FSUB` `FSUBR`, `FMUL`, `FDIV`, `FDIVR`, and `FSQRT` instructions. Lowering the precision will **not** decrease the execution time of FP instructions.

The `MCW_PC` field is set to use the full-precision 64-bit mantissa by the DJGPP startup code.

The rounding-control field `MCW_RC` (bits 10 and 11) controls the type (round or chop) and direction (-Inf or +Inf) of the rounding. It only affects arithmetic instructions. Set to round-to-nearest state by the DJGPP startup code.

The infinity-control bit `MCW_IC` has no effect on 80387 and later coprocessors.

Return Value

The previous control word.

(Note that this is different from what `_control87` from the Borland C library which returns the *new* control word.)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* mask all exceptions, except invalid operation */
_control87 (0x033e, 0xffff);
```

cos

Syntax

```
#include <math.h>

double cos(double x);
```

Description

This function computes the cosine of x (which should be given in radians).

Return Value

The cosine of x. If the absolute value of x is finite but greater than or equal to 2^63, the value is 1 (since for arguments that large each bit of the mantissa is more than `Pi`). If the value of x is infinite or `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Accuracy

In general, this function's relative accuracy is about 1.7*10^(-16), which is close to the machine precision for a `double`. However, for arguments very close to `Pi/2` and its odd multiples, the relative accuracy can be many times worse, due to loss of precision in the internal FPU computations. Since cos(Pi/2) is zero, the absolute accuracy is still very good; but if your program needs to preserve high *relative* accuracy for such arguments, link with `-lm` and use the version of `cos` from `libm.a` which does elaborate argument reduction, but is about three times slower.

cosh

Syntax

```
#include <math.h>

double cosh(double x);
```

Description

This function computes the hyperbolic cosine of x.

Return Value

The hyperbolic cosine of x. If the value of x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`. If the value of x is so large that the result would overflow a `double`, the return value is `Inf` and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is `+Inf`, and `errno` is not changed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

cprintf

Syntax

```
#include <conio.h>

int cprintf(const char *_format, ...);
```

Description

Like `printf` (See printf), but prints through the console, taking into consideration window borders and text attributes. There is currently a 2048-byte limit on the size of each individual cprintf call.

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

cputs

Syntax

```
#include <conio.h>

int cputs(const char *_str);
```

Description

Puts the string onto the console. The cursor position is updated.

Return Value

Zero on success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

creat

Syntax

```
#include <fcntl.h>
#include <sys/stat.h> /* for mode definitions */

int creat(const char *filename, mode_t mode);
```

Description

This function creates the given file and opens it for writing. If the file exists, it is truncated to zero size, unless it is read-only, in which case the function fails. If the file does not exist, it will be created read-only if mode does not have `S_IWUSR` set.

Return Value

A file descriptor >= 0, or a negative number on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int fd = creat("data", S_IRUSR|S_IWUSR);
write(fd, buf, 1024);
close(fd);
```

_creat

Syntax

```
#include <io.h>

int _creat(const char *path, int attrib);
```

Description

This is a direct connection to the MS-DOS creat function call, int 0x21, %ah = 0x3c, on versions of DOS earlier than 7.0. On DOS version 7.0 or later `_creat` calls function int 0x21, %ax = 0x6c00

On platforms where the LFN API (See _use_lfn, LFN) is available, `_creat` calls function 0x716C of Interrupt 21h, to support long file names.

On FAT32 file systems file sizes up to 2^32-2 are supported. Note that WINDOWS 98 has a bug which only lets you create these big files if LFN is enabled. In plain DOS mode it plainly works.

The file is set to binary mode.

This function can be hooked by File System Extensions (See File System Extensions). If you don't want this, you should use `_dos_creat` (See _dos_creat) or `_dos_creatnew` (See _dos_creatnew).

Return Value

The new file descriptor, else -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_creatnew

Syntax

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

int _creatnew(const char *path, int attrib, int flags);
```

Description

This function creates a file given by path and opens it, like `_creat` does, but only if it didn't already exist. If the named file exists, `_creatnew` fails. (In contrast, `_creat` opens existing files and overwrites their contents, see See _creat.)

The attributes of the created file are determined by attrib. The file is usually given the normal attribute (00H). If attrib is non-zero, additional attributes will be set. The following macros, defined on `<dir.h>`, can be used to control the attributes of the created file (the associated numeric values appear in parentheses):

`FA_RDONLY (1)`
: The file is created with the read-only bit set.

`FA_HIDDEN (2)`
: The file is created with the hidden bit set. Such files will not appear in directory listings unless you use special options to the commands which list files.

`FA_SYSTEM (4)`
: The file is created with the system bit set. Such files will not appear in directory listings unless you use special options to the commands which list files.

Other bits (`FA_LABEL` and `FA_DIREC`) are ignored by DOS.

The argument flags controls the sharing mode and the fine details of how the file is handled by the operating system. The following macros, defined on `<fcntl.h>`, can be used for this (associated numeric values are given in parentheses):

`SH_COMPAT (00h)`
: Opens the file in compatibility mode, which allows any other process to open the file and read from the file any number of times.

`SH_DENYRW (10h)`
: Denies both read and write access by other processes.

`SH_DENYWR (20h)`
: Denies write access by other processes.

`SH_DENYRD (30h)`
: Denies read access by other processes.

`SH_DENYNO (40h)`
: Allows read and write access by other processes, but prevents other processes from opening the file in compatibility mode.

Note that the file is always open for both reading and writing; `_creatnew` ignores any bits in the lower nibble of flags (`O_RDONLY`, `O_WRONLY`, etc.).

`_creatnew` calls DOS function 716Ch when long file names are supported, 6C00h otherwise. (On DOS version 3.x, function 5B00h is called which ignores the value of flags, since function 6C00h is only supported by DOS 4.0 and later.)

The file handle returned by `_creatnew` is set to binary mode.

This function can be hooked by the Filesystem Extensions handlers, as described in See File System Extensions. If you don't want this, you should use `_dos_creatnew` (See _dos_creatnew) instead.

Return Value

The new file descriptor, else -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

crlf2nl

Syntax

```
#include <io.h>

size_t crlf2nl(char *buf, ssize_t len);
```

Description

This function removes Ctrl-M characters from the given buf.

Return Value

The number of characters remaining in the buffer are returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__crt0_glob_function

Syntax

```
#include <crt0.h>

char **__crt0_glob_function(char *_argument);
```

Description

If the application wishes to provide a wildcard expansion function, it should define a `__crt0_glob_function` function. It should return a list of the expanded values, or 0 if no expansion will occur. The startup code will free the returned pointer if it is nonzero.

If no expander function is provided, wildcards will be expanded in the POSIX.1 style by the default `__crt0_glob_function` from the C library. To disable expansion, provide a `__crt0_glob_function` that always returns 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__crt0_load_environment_file

Syntax

```
#include <crt0.h>

void __crt0_load_environment_file(char *_app_name);
```

Description

This function, provided by libc.a, does all the work required to load additional environment variables from the file `djgpp.env` whose full pathname is given by the `DJGPP` environment variable. If the application does not use environment variables, the programmer can reduce the size of the program image by providing a version of this function that does nothing.

See __crt0_setup_arguments.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__crt0_setup_arguments

Syntax

```
#include <crt0.h>

void __crt0_setup_arguments(void);
```

Description

This function, provided by libc.a, does all the work required to provide the two arguments passed to main() (usually `argc` and `argv`). If main() does not use these arguments, the programmer can reduce the size of the program image by providing a version of this function that does nothing.

Note that since the default `__crt0_setup_arguments_function` will *not* expand wildcards inside quotes (**"** or **'**), you can quote a part of the argument that doesn't include wildcards and still have them expanded. This is so you could use wildcard expansion with filenames which have embedded whitespace (on LFN filesystems).

See __crt0_load_environment_file.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_crt0_startup_flags

Syntax

```
#include <crt0.h>

int _crt0_startup_flags = ...;
```

Description

This variable can be used to determine what the startup code will (or will not) do when the program begins. This can be used to tailor the startup environment to a particular program.

`_CRT0_FLAG_PRESERVE_UPPER_CASE`
: If set, `argv[0]` is left in whatever case it was. If not set, all characters are mapped to lower case. Note that if the `argv0` field in the stubinfo structure is present, the case of that part of `argv[0]` is not affected.

`_CRT0_FLAG_USE_DOS_SLASHES`
: If set, reverse slashes (dos-style) are preserved in `argv[0]`. If not set, all reverse slashes are replaced with unix-style slashes.

`_CRT0_FLAG_DROP_EXE_SUFFIX`
: If set, the `.exe` suffix is removed from the file name component of `argv[0]`. If not set, the suffix remains.

`_CRT0_FLAG_DROP_DRIVE_SPECIFIER`
: If set, the drive specifier (e.g. `C:`) is removed from the beginning of `argv[0]` (if present). If not set, the drive specifier remains.

`_CRT0_FLAG_DISALLOW_RESPONSE_FILES`
: If set, response files (e.g. `@gcc.rf`) are not expanded. If not set, the contents of the response files are used to create arguments. Note that if the file does not exist, that argument remains unexpanded.

`_CRT0_FLAG_KEEP_QUOTES`
: If set, the quote characters `'`, `"`, and `\` will be retained in `argv[]` elements when processing command lines passed by DOS and via `system`. This is used by the `redir` program, and should only be needed if you want to get the original command line exactly as it was passed by the caller.

`_CRT0_FLAG_FILL_SBRK_MEMORY`
: If set, fill `sbrk`'d memory with a constant value. If not, memory gets whatever happens to have been in there, which breaks some applications.

`_CRT0_FLAG_FILL_DEADBEEF`
: If set, fill memory (above) with `0xdeadbeef`, else fill with zero. This is especially useful for debugging uninitialized memory problems.

`_CRT0_FLAG_NEARPTR`
: If set, set DS limit to 4GB which allows use of near pointers to DOS (and other) memory. WARNING, disables memory protection and bad pointers may crash the machine or wipe out your data. This flag is silently ignored on NT and DOSEmu, which disallow such huge selector limits.

`_CRT0_FLAG_NULLOK`
: If set, disable NULL pointer protection (if it can be controlled at all).

`_CRT0_FLAG_NMI_SIGNAL`
: If set, enabled capture of NMI in exception code. This may cause problems with laptops and "green" boxes which use it to wake up. Default is to leave NMIs alone and pass through to real mode code. You decide.

`_CRT0_FLAG_NO_LFN`
: If set, disable usage of long file name functions even on systems (such as Windows 9X) which support them. This might be needed to work around program assumptions on file name format on programs written specifically for DOS. Note that this flag overrides the value of the environment variable `LFN`.

`_CRT0_FLAG_NONMOVE_SBRK`
: If set, the `sbrk` algorithm uses multiple DPMI memory blocks which makes sure the base of CS/DS/SS does not change. This may cause problems with `sbrk(0)` values and programs with other assumptions about `sbrk` behavior. This flag is useful with near pointers, since a constant pointer to DOS/Video memory can be computed without needing to reload it after any routine which might call `sbrk`.

`_CRT0_FLAG_UNIX_SBRK`
: If set, the `sbrk` algorithm resizes memory blocks so that the layout of memory is set up to be the most compatible with Unix `sbrk` expectations. This mode should not be used with hardware interrupts, near pointers, and may cause problems with QDPMI virtual memory.

  If your program requires a specific `sbrk` behavior, you should set either this or the previous flag, since the default may change in different libc releases.

`_CRT0_DISABLE_SBRK_ADDRESS_WRAP`
: If set, non-move `sbrk` should discard (ignore) memory blocks which are returned by DPMI which would require address wrap to access (at addresses below the CS/DS base address). This bit is automatically set on Windows NT systems which require it. It may be manually set on other systems which don't require it to retain a more normal memory space layout and better memory protection. This bit can be set but should never be cleared.

`_CRT0_FLAG_LOCK_MEMORY`
: If set, locks all memory as it is allocated. This effectively disables virtual memory, and may be useful if using extensive hardware interrupt codes in a relatively small image size. The memory is locked after it is `sbrk`'ed, so the locking may fail. This bit may be set or cleared during execution. When `sbrk` uses multiple memory zones, it can be difficult to lock all memory since the memory block size and location is impossible to determine.

`_CRT0_FLAG_PRESERVE_FILENAME_CASE`
: If set, disables all filename letter-case conversions in functions that traverse directories (except findfirst/findnext which always return the filenames exactly as found in the directory entry). When reset, all filenames on 8+3 MSDOS filesystems and DOS-style 8+3 filenames on LFN systems are converted to lower-case by functions such as 'readdir', `getcwd`, `_fixpath` and others. Note that when this flag is set, ALL filenames on MSDOS systems will appear in upper-case, which is both ugly and will break many Unix-born programs. Use only if you know exactly what you are doing!

  This flag overrides the value of the environment variable `FNCASE`, See _preserve_fncase.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

cscanf

Syntax

```
#include <conio.h>

int cscanf(const char *_format, ...);
```

Description

Like `scanf` (See scanf), but it reads from the standard input device directly, avoiding buffering both by DOS and by the library. Each character is read by `getche` (See getche).

Return Value

The number of fields stored.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

ctermid

Syntax

```
#include <unistd.h>

char *ctermid(char *s);
```

Description

This function returns the name of the current terminal device. Under MS-DOS, this is always "con".

Return Value

If s is null, returns pointer to internal static string "con". Otherwise, copies "con" to buffer pointed by s.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

ctime

Syntax

```
#include <time.h>

char *ctime(const time_t *cal);
```

Description

This function returns an ASCII representation of the time in cal. This is equivalent to `asctime(localtime(cal))`. See asctime. See localtime.

Return Value

The ascii representation of the time.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

delay

Syntax

```
#include <dos.h>

void delay(unsigned msec);
```

Description

This function causes the program to pause for msec milliseconds. It uses the `int 15h` delay function to relinquish the CPU to other programs that might need it.

Some operating systems that emulate DOS, such as OS/2, Windows/NT, Windows 2000 and Windows XP hang the DOS session when the **Pause** key is pressed during the call to `delay`. Plain DOS and Windows 3.X and 9X are known to not have this bug. On Windows 2000 and XP to exit the pause press any key.

Some operating systems, such as Windows 2000 and XP which do not support `int 15h`. `int 1ah` is used instead on these operating systems. This method has lower accuracy in the delay length.

Windows 2000 and XP delay resolution is 54.9 millisecond. Under Windows 2000 and XP the delay function uses the Time Of Day Tick which occurs 18.2 times per second. This limits the accuracy of the delay to around 27 milliseconds on Windows 2000 and XP. On Windows 2000 and XP the Programable Interval Timer works and is a source of higher resolution than delay currently uses. Unfortunately PIT and Time Of Day tic does not appear to be coordinated.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
delay(200); /* delay for 1/5 second */
```

delline

Syntax

```
#include <conio.h>

void delline(void);
```

Description

The line the cursor is on is deleted; lines below it scroll up.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It’s not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don’t have any control on the order in which static constructors are called (it’s entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

_detect_80387

Syntax

```
#include <dos.h>

int _detect_80387(void);
```

Description

Detects whether a numeric coprocessor is present. Note that floating-point code will work even without a coprocessor, due to the existence of emulation.

Return Value

1 if a coprocessor is present, 0 if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (_detect_80387())
printf("You have a coprocessor\n");
```

difftime

Syntax

```
#include <time.h>

double difftime(time_t t1, time_t t0);
```

Description

This function returns the difference in time, in seconds, from t0 to t1.

Return Value

The number of seconds.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
time_t t1, t0;
double elapsed;
time(&t0);
do_something();
time(&t1);
elapsed = difftime(t1, t0);
```

dirname

Syntax

```
#include <unistd.h>

char * dirname (const char *fname);
```

Description

This function returns the directory part of the argument fname copied to a buffer allocated by calling `malloc`. The directory part is everything up to but not including the rightmost slash (either forward- or backslash) in fname. If fname includes a drive letter but no slashes, the function will return `x:.` where x is the drive letter. If fname includes neither the drive letter nor any slashes, `"."` will be returned. Trailing slashes are removed from the result, unless it is a root directory, with or without a drive letter.

Return value

The directory part in malloc'ed storage, or a NULL pointer of either there's not enough free memory, or fname is a NULL pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf ("The parent of current directory is %s\n",
dirname (getcwd (0, PATH_MAX)));
```

disable

Syntax

```
#include <dos.h>

int disable(void);
```

Description

This function disables interrupts.

See enable.

Return Value

Returns nonzero if the interrupts had been enabled before this call, zero if they were already disabled.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int ints_were_enabled;

ints_were_enabled = disable();
. . . do some stuff . . .
if (ints_were_enabled)
enable();
```

div

Syntax

```
#include <stdlib.h>

div_t div(int numerator, int denominator);
```

Description

Returns the quotient and remainder of the division numerator divided by denominator. The return type is as follows:

```
typedef struct {
int quot;
int rem;
} div_t;
```

Return Value

The results of the division are returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
div_t d = div(42, 3);
printf("42 = %d x 3 + %d\n", d.quot, d.rem);

div(+40, +3) = { +13, +1 }
div(+40, -3) = { -13, +1 }
div(-40, +3) = { -13, -1 }
div(-40, -3) = { +13, -1 }
```

__djgpp_exception_toggle

Syntax

```
#include <sys/exceptn.h>

void __djgpp_exception_toggle(void);
```

Description

This function is automatically called when the program exits, to restore handling of all the exceptions to their normal state. You may also call it from your program, around the code fragments where you need to temporarily restore **all** the exceptions to their default handling. One example of such case might be a call to a library function that spawns a child program, when you don't want to handle signals generated while the child runs (by default, those signals are also passed to the parent).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
__djgpp_exception_toggle();
system("myprog");
__djgpp_exception_toggle();
```

__djgpp_map_physical_memory

Syntax

```
#include <dpmi.h>

int __djgpp_map_physical_memory(void *our_addr, unsigned long num_bytes,
unsigned long phys_addr);
```

Description

This function attempts to map a range of physical memory over the specified addresses. One common use of this routine is to map device memory, such as a linear frame buffer, into the address space of the calling program. our_addr, num_bytes, and phys_addr must be page-aligned. If they are not page-aligned, `errno` will be set to `EINVAL` and the routine will fail.

This routine properly handles memory ranges that span multiple DPMI handles, while `__dpmi_map_device_in_memory_block` does not.

Consult DPMI documentation on function 0508H for details on how this function works. Note: since 0508H is a DPMI service new with DPMI 1.0, this call will fail on most DPMI 0.9 servers. For your program to work on a wide range of systems, you should not assume this call will succeed.

Even on failure, this routine may affect a subset of the pages specified.

Return Value

0 on success, -1 on failure. On failure, `errno` will be set to `EINVAL` for illegal input parameters, or `EACCES` if the DPMI server rejected the mapping request.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (__djgpp_map_physical_memory (my_page_aligned_memory, 16384,
0x40000000))
printf ("Failed to map physical addresses!\n");
```

__djgpp_memory_handle

Syntax

```
#include <crt0.h>

__djgpp_sbrk_handle *__djgpp_memory_handle(unsigned address);
```

Description

This function returns a pointer to a structure containing the memory handle and program relative offset associated with the address passed. It is just a convenient way to process the __djgpp_memory_handle_list.

Return Value

A pointer to the __djgpp_sbrk_handle associated with a particular address.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__djgpp_memory_handle_list

Syntax

```
#include <crt0.h>

extern __djgpp_sbrk_handle __djgpp_memory_handle_list[256];
```

Description

This array contains a list of memory handles and program relative offsets allocated by sbrk() in addition to the handle allocated by the stub. These values are normally not needed unless you are doing low-level DPMI page protection or memory mapping.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <crt0.h>

for(i=0; i<256; i++) {
int h, a, s;
h = __djgpp_memory_handle_list[i].handle;
a = __djgpp_memory_handle_list[i].address;
s = __djgpp_memory_handle_size[i];
if(a == 0 && i != 0) break;
printf("handle[%d]=0x%x base=0x%x size=0x%x\n",i,h,a,s);
}
```

__djgpp_memory_handle_size

Syntax

```
#include <crt0.h>

extern unsigned __djgpp_memory_handle_size[256];
```

Description

This array contains a list of the sizes of the memory regions allocated by sbrk() in addition to the memory region allocated by the stub. These values are normally not needed unless you are dumping the memory blocks.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <crt0.h>

for(i=0; i<256; i++) {
int h, a, s;
h = __djgpp_memory_handle_list[i].handle;
a = __djgpp_memory_handle_list[i].address;
s = __djgpp_memory_handle_size[i];
if(a == 0 && i != 0) break;
printf("handle[%d]=0x%x base=0x%x size=0x%x\n",i,h,a,s);
}
```

__djgpp_nearptr_disable

Syntax

```
#include <sys/nearptr.h>

void __djgpp_nearptr_disable(void);
```

Description

This function disables near pointers, and re-enables protection. See __djgpp_nearptr_enable.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__djgpp_nearptr_enable

Syntax

```
#include <sys/nearptr.h>

int __djgpp_nearptr_enable(void);
```

Description

This function enables "near pointers" to be used to access the DOS memory arena. Sort of. When you call this function, it will return nonzero if it has successfully enabled near pointers. If so, you must add the value `__djgpp_conventional_base` to the linear address of the physical memory. For example:

```
if (__djgpp_nearptr_enable())
{

short *screen = (short *)(__djgpp_conventional_base + 0xb8000);
for (i=0; i<80*24*2; i++)
screen[i] = 0x0720;
__djgpp_nearptr_disable();
}
```

The variable `__djgpp_base_address` contains the linear base address of the application's data segment. You can subtract this value from other linear addresses that DPMI functions might return in order to obtain a near pointer to those linear regions as well.

If using the Unix-like sbrk algorithm, near pointers are only valid until the next `malloc`, `system`, `spawn*`, or `exec*` function call, since the linear base address of the application may be changed by these calls.

WARNING: When you enable near pointers, you disable all the protection that the system is providing. If you are not careful, your application may destroy the data in your computer. USE AT YOUR OWN RISK!

Return Value

Returns 0 if near pointers are not available, or nonzero if they are.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__djgpp_set_ctrl_c

Syntax

```
#include <sys/exceptn.h>

int __djgpp_set_ctrl_c(int enable);
```

Description

This function sets and resets the bit which controls whether signals `SIGINT` and `SIGQUIT` (See signal) will be raised when you press the INTR or QUIT keys. By default these generate signals which, if uncaught by a signal handler, will abort your program. However, when you call the `setmode` library function to switch the console reads to binary mode, or open the console in binary mode for reading, this generation of signals is turned off, because some programs want to get the `^C` and `^\` characters as any other character and handle them by themselves.

`__djgpp_set_ctrl_c` lets you explicitly determine the effect of INTR and QUIT keys. When called with a non-zero, positive value of enable, it arranges for `SIGINT` and `SIGQUIT` signals to be generated when the appropriate key is pressed; if you call it with a zero in enable, these keys are treated as normal characters. If enable is negative, `__djgpp_set_ctrl_c` returns the current state of the signal generation, but doesn't change it.

For getting similar effects via the POSIX `termios` functions, see See tcsetattr.

Note that the effect of **Ctrl-BREAK** key is unaffected by this function; use the `_go32_want_ctrl_break` library function to control it.

Also note that in DJGPP, the effect of the interrupt signal will only be seen when the program is in protected mode (See signal, Signal Mechanism, for more details). Thus, if you press **Ctrl-C** while your program calls DOS (e.g., when reading from the console), the `SIGINT` signal handler will only be called after that call returns.

Return Value

The state of `SIGINT` and `SIGQUIT` generation before the call: 0 if it was disabled, 1 if it was enabled. If the argument enable is negative, the state is not altered.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
setmode(fileno(stdin), O_BINARY);
if (isatty(fileno(stdin)));
__djgpp_set_ctrl_c(1);
```

__djgpp_set_page_attributes

Syntax

```
#include <dpmi.h>

int __djgpp_set_page_attributes(void *our_addr, unsigned long num_bytes,
unsigned short attributes);
```

Description

This function sets the DPMI page attributes for the pages in a range of memory. our_addr and num_bytes must be page-aligned. If they are not page-aligned, `errno` will be set to `EINVAL` and the routine will fail.

Consult DPMI documentation on function 0507H for the meaning of the attributes argument. Note: since 0507H is a DPMI service new with DPMI 1.0, this call will fail on most DPMI 0.9 servers. For your program to work on a wide range of systems, you should not assume this call will succeed.

Even on failure, this routine may affect a subset of the pages specified.

Return Value

0 on success, -1 on failure. On failure, `errno` will be set to `EINVAL` for illegal input parameters, or `EACCES` if the DPMI server rejected the attribute setting.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (__djgpp_set_page_attributes (my_page_aligned_memory, 16384, 0))
printf ("Failed to make pages uncommitted!\n");
```

__djgpp_set_sigint_key

Syntax

```
#include <sys/exceptn.h>

void __djgpp_set_sigint_key(int new_key);
```

Description

This function changes the INTR key that generates the signal `SIGINT`. By default, **Ctrl-C** is set as the INTR key. To replace it with another key, put the *scan code* of the new INTR key into the bits 0-7 and the required keyboard status byte into bits 8-15 of new_key, and call this function. Here's how the keyboard status bits are defined:

```
Bit
76543210 Meaning

.......X Right Shift key
......X. Left Shift key
.....X.. Ctrl key
....X... Alt key
...X.... Scroll Lock key
..X..... Num Lock key
.X...... Caps Lock key
X....... Insert
```

A 1 in any of the above bits means that the corresponding key should be pressed; a zero means it should be released. Currently, all but the lower 4 bits are always ignored by the DJGPP keyboard handler when you set the INTR key using this function.

For example, the default **Ctrl-C** key should be passed as `0x042e`, since the scan code of the **C** key is 2Eh, and when the **Ctrl** key is pressed, the keyboard status byte is 04h.

To disable `SIGINT` generation, pass zero as the argument (since no key has a zero scan code).

This function will set things up so that the left **Shift** key doesn't affect Ctrl- and Alt-modified keys; the right **Shift** key won't affect them either, unless its bit is explicitly set in new_key. This means that **Ctrl-C** and **Ctrl-c** will both trigger `SIGINT` if `0x042e` is passed to this function.

The DJGPP built-in keyboard handler pretends that when the right **Shift** key is pressed, so is the left **Shift** key (but not vice versa).

For getting similar effects via the POSIX `termios` functions, see See tcsetattr.

Return Value

The previous INTR key (scan code in bits 0-7, keyboad status in bits 8-15).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
__djgpp_set_sigint_key(0x0422); /* make Ctrl-g generate SIGINT's */
```

__djgpp_set_sigquit_key

Syntax

```
#include <sys/exceptn.h>

void __djgpp_set_sigquit_key(int new_key);
```

Description

This function changes the QUIT key that generates the signal `SIGQUIT`. By default, **Ctrl-** is set as the QUIT key. To replace it with another key, put the *scan code* of the new QUIT key into the bits 0-7 and the required keyboard status byte into bits 8-15 of new_key, and call this function. Here's how the keyboard status bits are defined:

```
Bit
76543210 Meaning

.......X Right Shift key
......X. Left Shift key
.....X.. Ctrl key
....X... Alt key
...X.... Scroll Lock key
..X..... Num Lock key
.X...... Caps Lock key
X....... Insert
```

A 1 in any of the above bits means that the corresponding key should be pressed; a zero means it should be released. Currently, all but the lower 4 bits are always ignored by the DJGPP keyboard handler when you set the QUIT key with this function.

For example, the default **Ctrl-** key should be passed as `0x042b`, since the scan code of \ is 2Bh and when the **Ctrl** key is pressed, the keyboard status byte is 04h.

To disable `SIGQUIT` generation, pass zero as the argument (since no key has a zero scan code).

This function will set things up so that the left **Shift** key doesn't affect Ctrl- and Alt-modified keys; the right **Shift** key won't affect them either, unless its bit is explicitly set in new_key. This means that **Ctrl-** and **Ctrl-|** will both trigger `SIGQUIT` if `0x042b` is passed to this function.

The DJGPP built-in keyboard handler pretends that when the right **Shift** key is pressed, so is the left **Shift** key (but not vice versa).

For getting similar effects via the POSIX `termios` functions, see See tcsetattr.

Return Value

The previous QUIT key (scan code in bits 0-7, keyboad status in bits 8-15).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
__djgpp_set_sigint_key(0); /* disable SIGQUIT's */
```

__djgpp_share_flags

Syntax

```
#include <fcntl.h>

int __djgpp_share_flags = ...;
```

Description

This variable controls the share flags used by `open` (and hence `fopen`) when opening a file.

If you assign any value other than 0 to this variable libc will use that value for the sharing bits when if calls DOS to open the file. But if you specify any share flag in the `open` call then these flags will remain untouched. In this way `__djgpp_share_flags` acts just like a default and by default is 0 ensuring maximum compatibility with older versions of djgpp.

If you don't know how the share flags act consult any DOS reference. They allow to share or protect a file when it's opened more than once by the same task or by two or more tasks. The exact behavior depends on the exact case. One interesting thing is that when the file is opened by two tasks under Windows the results are different if you use Windows 3.1 or Windows 95. To add even more complexity Windows 3.1 is affected by `SHARE.EXE`.

The available flags are:

`SH_COMPAT 0x0000`
: That's the compatible mode.

`SH_DENYRW 0x0010`
: Deny read and deny write.

`SH_DENYWR 0x0020`
: Deny write.

`SH_DENYRD 0x0030`
: Deny read.

`SH_DENYNO 0x0040`
: No deny.

Of course these flags are DOS specific and doesn't exist under other OSs; and as you can imagine `__djgpp_share_flags` is djgpp specific.

See open. See fopen.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__djgpp_spawn

Syntax

```
#include <process.h>

int __djgpp_spawn(int mode, const char *path, char *const argv[],
char *const envp[], unsigned long flags);
```

Description

This function runs other programs like `spawnve` (See spawn*) except that an additional parameter flags is passed. flags can include the following flags to control the details of finding the program to run:

`SPAWN_EXTENSION_SRCH`
: If an extension is not included in path, search for a file path with the extensions `.com`, `.exe`, `.bat`, and `.btm`.

`SPAWN_NO_EXTENSION_SRCH`
: Do not perform an extension search. If the file has an extension, it must already be included in path.

Return Value

See spawn*.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *args[] = {
"gcc.exe",
"-v",
"hello.c",
0
};

__djgpp_spawn(P_WAIT, "/dev/env/DJDIR/bin/gcc.exe", args, NULL,
SPAWN_NO_EXTENSION_SRCH);
```

__djgpp_traceback_exit

Syntax

```
#include <signal.h>

void __djgpp_traceback_exit(int signo);
```

Description

This function is a signal handler which will print a traceback and abort the program. It is called by default by the DJGPP signal-handling code when any signal except `SIGQUIT` is raised (`SIGQUIT` is by default discarded).

You can use this function to get the Unix behavior of aborting the program on `SIGQUIT` (see the example below).

When this function is called directly, pass the signal number as its signo argument.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
signal(SIGQUIT, __djgpp_traceback_exit);
```

_djstat_describe_lossage

Syntax

```
#include <stdio.h>

void _djstat_describe_lossage(FILE *fp);
```

Description

Accesses the global variable `_djstat_fail_bits` (See _djstat_fail_bits) and prints to the stream given by fp a human-readable description of the undocumented DOS features which the last call to `stat()` or `fstat()` failed to use. (If fp is zero, the function prints to stderr.) If the last call to `f?stat()` didn't set any failure bits, an ''all's well'' message is printed. This function is designed to help in debugging these functions in hostile environments (like DOS clones) and in adapting them to the future DOS versions. If you ever have any strange results returned by `f?stat()`, please call this function and post the diagnostics it printed to the DJGPP mailing list.

The diagnostic messages this function prints are almost self-explanatory. Some explanations of terminology and abbreviations used by the printed messages will further clarify them.

SDA (Swappable DOS Area) -- this is an internal DOS structure. `stat()` uses it to get the full directory entry (including the starting cluster number) of a file. The pointer to SDA found by `stat()` is trusted only if we find the pathname of our file at a specific offset in that SDA.

SFT (System File Table) -- another internal DOS structure, used in file operations. `fstat()` uses it to get full information on a file given its handle. An SFT entry which is found by `fstat()` is only trusted if it contains files size and time stamp like those returned by DOS functions 57h and 42h. Novell NetWare 3.x traps DOS file operations in such a way they never get to SFT, so some failure messages refer specifically to Novell.

Hashing -- the fall-back method of returning a unique inode number for each file. It is used whenever the starting cluster of a file couldn't be reliably determined.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (stat(path, &stat_buf))
_djstat_describe_lossage((FILE *)0);
```

_djstat_fail_bits

Syntax

```
#include <sys/stat.h>

extern unsigned short _djstat_fail_bits;
```

As proper operation of `stat` (See stat) and `fstat` (See fstat) depend on undocumented DOS features, they could fail in some incompatible environment or a future DOS version. If they do, the `_djstat_fail_bits` variable will have some of its bits set. Each bit describes a single feature which was used and failed. The function `_djstat_describe_lossage` (See _djstat_describe_lossage) may be called to print a human-readable description of the bits which were set by the last call to `f?stat`. This should make debugging `f?stat` failures in an unanticipated environment a lot easier.

The following bits are currently defined:

`_STFAIL_SDA`
: Indicates that Get SDA call failed.

`_STFAIL_OSVER`
: Indicates an unsupported DOS version (less than 3.10 for `stat` or less than 2.0 for `fstat`).

`_STFAIL_BADSDA`
: The pointer to SDA was found to be bogus.

`_STFAIL_TRUENAME`
: Indicates that `_truename` (See _truename) function call failed.

`_STFAIL_HASH`
: Indicates that the starting cluster of the file is unavailable, and inode number was computed by hashing its name.

`_STFAIL_LABEL`
: The application requested the time stamp of a root dir, but no volume label was found.

`_STFAIL_DCOUNT`
: The number of SDA reported is ridiculously large (probably an unsupported DOS clone).

`_STFAIL_WRITEBIT`
: `fstat` was asked to get write access bit of a file, but couldn't.

`_STFAIL_DEVNO`
: `fstat` failed to get device number.

`_STFAIL_BADSFT`
: An SFT entry for this file was found by `fstat`, but its contents can't be trusted because it didn't match file size and time stamp as reported by DOS.

`_STFAIL_SFTIDX`
: The SFT index in Job File Table in program's PSP is negative.

`_STFAIL_SFTNF`
: The file entry was not found in the SFT array.

Below are some explanations of terminology and abbreviations used by the printed messages, which will further clarify the meaning of the above bits and their descriptions printed by `_djstat_describe_lossage` (See _djstat_describe_lossage).

SDA (Swappable Data Area) -- this is an internal DOS structure. `stat` uses it to get the full directory entry (including the starting cluster number) of a file. The pointer to SDA found by `stat` is trusted only if we find the pathname of our file at a specific offset in that SDA.

SFT (System File Table) -- another internal DOS structure, used in file operations. `fstat` uses it to get full information on a file given its handle. An SFT entry which is found by `fstat` is only trusted if it contains files size and time stamp like those returned by DOS functions 57h and 42h. Novell NetWare 3.x traps DOS file operations in such a way they never get to SFT, so some failure messages refer specifically to Novell.

Hashing -- the fall-back method of returning a unique inode number for each file. It is used whenever the starting cluster of a file couldn't be reliably determined. The full pathname of the file is looked up in a table of files seen earlier (hashing is used to speed the lookup process). If found, the inode from the table is returned; this ensures that a given file will get the same inode number. Otherwise a new inode number is invented, recorded in the table and returned to caller.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_djstat_flags

Syntax

```
#include <sys/stat.h>

extern unsigned short _djstat_flags;
```

This variable contains bits for some fields of `struct stat` which are expensive to compute under DOS. Any such computation is only done by `stat` (See stat) or `fstat` (See fstat) if the corresponding bit in `_djstat_flags` is *cleared*. By default, all the bits are cleared, so applications which don't care, automagically get a full version, possibly at a price of performance. To get the fastest possible version for your application, clear only the bits which you need and set all the others.

The following bits are currently defined:

`_STAT_INODE`
: Causes `stat` and `fstat` to compute the `st_ino` (inode number) field.

`_STAT_EXEC_EXT`
: Tells `stat` and `fstat` to compute the execute access bit from the file-name extension. `stat` and `fstat` know about many popular file-name extensions, to speed up the computation of the execute access bit.

`_STAT_EXEC_MAGIC`
: Tells `stat` and `fstat` to compute the execute access bit from magic signature (the first two bytes of the file). Use `_is_executable` (See _is_executable), if the file-name extension is not enough for this.

  Computing the execute access bit from the magic signature is by far the most expensive part of `stat` and `fstat` (because it requires to read the first two bytes of every file). If your application doesn't care about execute access bit, setting `_STAT_EXEC_MAGIC` will significantly speed it up.

  Note that if `_STAT_EXEC_MAGIC` is set, but `_STAT_EXEC_EXT` is not, some files which shouldn't be flagged as executables (e.g., COFF `*.o` object files) will have their execute bit set, because they have the magic number signature at their beginning. Therefore, only use the above combination if you want to debug the list of extensions provided in `is_exec.c` file.

`_STAT_DIRSIZE`
: Causes `stat` to compute directory size by counting the number of its entries (unless some friendly network redirector brought a true directory size with it). Also computes the number of subdirectories and sets the number of links `st_nlink` field.

  This computation is also quite expensive, especially for directories with large sub-directories. If your application doesn't care about size of directories and the `st_nlink` member, you should set the `_STAT_DIRSIZE` bit in `_djstat_flags`.

`_STAT_ROOT_TIME`
: Causes `stat` to try to get time stamp of root directory from its volume label entry, if there is one.

`_STAT_WRITEBIT`
: Tells `fstat` that file's write access bit is required (this needs special treatment only under some versions of Novell Netware).

Note that if you set a bit, some failure bits in `_djstat_fail_bits` (See _djstat_fail_bits) might not be set, because some computations which report failures are only done when they are required.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

dlclose

Syntax

```
#include <dlfcn.h>

int dlclose (void *handle);
```

Description

This function closes a dynamic module loaded with `dlopen` (See dlopen). The memory is freed and all pointers into that image become invalid.

Return Value

Returns 0 on success, non-zero value on failure. More detailed error information can be obtained using `dlerror` (See dlerror).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.1-2001; not 1003.2-1992

dlerror

Syntax

```
#include <dlfcn.h>

char *dlerror (void);
```

Description

This function returns a character string with more information on the last error that occurred during dynamic linking processing.

Return Value

Returns `NULL` if no errors, pointer to error string if available.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.1-2001; not 1003.2-1992

dlerrstatmod

Syntax

```
#include <sys/dxe.h>

extern void (*dlerrstatmod) (const char *module);
```

Description

This is a pointer to a function (e.g. replaceable) containing a pointer to a function that is called when static linking fails because of missing module. Note that due to delayed nature of static linkage, the error can pop up very late! If you want to check it at startup, call the "load_MODULENAME" function explicitly. The function should never return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

dlerrstatsym

Syntax

```
#include <sys/dxe.h>

extern void (*dlerrstatsym) (const char *module, const char *symbol);
```

Description

This is a pointer to a function that is being called during static linking when the dynamic loader finds that some symbol is missing from dynamic module. The function should never return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

dlopen

Syntax

```
#include <dlfcn.h>

void *dlopen (const char *filename, int mode);
```

Description

This function loads a dynamic executable image, whose file name is pointed to by filename, into memory and returns a handle associated with the image for use with the `dlsym` (See dlsym) and `dlclose` (See dlclose) functions.

If filename contains a path it is used, else it searches the path specified by the environment variable LD_LIBRARY_PATH. The typical extension used is `.DXE`, and these dynamic loadable images are created using dxe3gen (See dxe3gen, , dxe3gen, utils).

The mode field is a combination of `RTLD_xxx` flags, of which only `RTLD_GLOBAL` works (others are defined in `dlfcn.h` for Unix compatibility). The `RDLD_GLOBAL` flag means all symbols in this module are made public and subsequently loaded modules with unresolved symbols will 'see' them and will try to find the unresolved references through them.

Return Value

`NULL` on failure, handle for the loaded image on success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.1-2001; not 1003.2-1992

dlregsym

Syntax

```
#include <sys/dxe.h>
```

```
int dlregsym (const dxe_symbol_table *symtab);
```

Description

This function registers a table of symbols to be exported into the loaded modules.

You can register any number of such tables. When a module with unresolved external symbols is loaded, all these tables are searched for the respective symbol. If none is found, the last-resort handler is called. If even the last-resort handler cannot return an address, an error condition is raised.

The effect of `dlregsym` is cumulative. That is, you can call it multiple times to register several export symbol tables, and all of them will be taken into account when loading a new module.

Return Value

Returns number of symbol tables in use.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

dlsym

Syntax

```
#include <dlfcn.h>

void *dlsym (void *handle, const char *symbol_name);
```

Description

This function get the address of a symbol defined in a shared loadable image.

The handle argument is the value returned from a call to `dlopen`, or the special value `RTLD_DEFAULT` which will search all symbols in the global scope.

The symbol_name is the assembler name, not the C name. For DJGPP/COFF prepend an underscore in front.

Return Value

Returns `NULL` on failure, pointer to requested symbol on success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.1-2001; not 1003.2-1992

dlsymresolver

Syntax

```
#include <sys/dxe.h>

extern void *(*dlsymresolver) (const char *symname);
```

Description

This is a pointer to a function (e.g. replaceable) containing a pointer to a function that is called when an unresolved symbol cannot be found in all the symbol tables that the dynamic loader have at his disposition. For example, as a last resort, the routine could return the address of a dummy function -- this allows loading modules that you don't know in advance which unresolved symbols it contains. Of course, the functions that use this last-resort dummy function will be, most likely, unuseable but at least you may query the address of some table inside the module, for example, and process it somehow.

Return Value

The handler should return `NULL` to rise a error condition, otherwise it should return a valid address.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

dlunregsym

Syntax

```
#include <sys/dxe.h>

int dlunregsym (const dxe_symbol_table *symtab);
```

Description

This function removes a table of symbols exported into the loaded modules. For completeness - the inverse of `dlregsym` and rarely used.

Return Value

Returns number of symbols tables in use if success, -1 on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_doprnt

Syntax

```
#include <stdio.h>

int _doprnt(const char *format, void *params, FILE *file);
```

Description

This is an internal function that is used by all the `printf` style functions, which simply pass their format, arguments, and stream to this function.

See printf, for a discussion of the allowed formats and arguments.

Return Value

The number of characters generated is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int args[] = { 1, 2, 3, 66 };
_doprnt("%d %d %d %c\n", args, stdout);
```

_dos_close

Syntax

```
#include <dos.h>

unsigned int _dos_close(int handle);
```

Description

This is a direct connection to the MS-DOS close function call (%ah = 0x3E). This function closes the specified file.

See _dos_open. See _dos_creat. See _dos_creatnew. See _dos_read. See _dos_write.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;

_dos_creat("FOO.DAT", _A_ARCH, &handle);
...
_dos_close(handle);
```

_dos_commit

Syntax

```
#include <dos.h>

unsigned int _dos_commit(int handle);
```

Description

This is a direct connection to the MS-DOS commit function call (%ah = 0x68). This function flushes DOS internal file buffers to disk.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_dos_write(handle, buffer, 1000, &result);
_dos_commit(handle);
_dos_close(handle);
```

_dos_creat

Syntax

```
#include <dos.h>

unsigned int _dos_creat(const char *filename, unsigned short attr,
int *handle);
```

Description

This is a direct connection to the MS-DOS creat function call (%ah = 0x3C). This function creates the given file with the given attribute and puts file handle into handle if creating is successful. If the file already exists it truncates the file to zero length. Meaning of attr parameter is the following:

`_A_NORMAL (0x00)`
: Normal file (no read/write restrictions)

`_A_RDONLY (0x01)`
: Read only file

`_A_HIDDEN (0x02)`
: Hidden file

`_A_SYSTEM (0x04)`
: System file

`_A_ARCH (0x20)`
: Archive file

See also See _dos_open, See _dos_creatnew, See _dos_read, See _dos_write, and See _dos_close.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See _creat, and See _creatnew. Also see See creat, and See open, which are Posix-standard.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno`)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;

if ( !_dos_creat("FOO.DAT", _A_ARCH, &handle) )
puts("Creating was successful !");
```

_dos_creatnew

Syntax

```
#include <dos.h>

unsigned int _dos_creatnew(const char *filename, unsigned short attr,
int *handle);
```

Description

This is a direct connection to the MS-DOS create unique function call (%ah = 0x5B). This function creates the given file with the given attribute and puts file handle into handle if creating is successful. This function will fail if the specified file exists. Meaning of attr parameter is the following:

`_A_NORMAL (0x00)`
Normal file (no read/write restrictions)

`_A_RDONLY (0x01)`
Read only file

`_A_HIDDEN (0x02)`
Hidden file

`_A_SYSTEM (0x04)`
System file

`_A_ARCH (0x20)`
Archive file

See also See _dos_open, See _dos_creat, See _dos_read, See _dos_write, and See _dos_close.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See _creatnew, and See _creat. Also see See creat, and See open, which are Posix-standard.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;

if ( !_dos_creatnew("FOO.DAT", _A_NORMAL, &handle) )
puts("Creating was successful !");
```

_dos_findfirst

Syntax

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

```
unsigned int _dos_findfirst(char *name, unsigned int attr,
struct find_t *result);
```

Description

This function and the related `_dos_findnext` (See _dos_findnext) are used to scan directories for the list of files therein. The name is a wildcard that specifies the directory and files to search. result is a structure to hold the results and state of the search, and attr is a combination of the following:

`_A_NORMAL (0x00)`
: Normal file (no read/write restrictions)

`_A_RDONLY (0x01)`
: Read only file

`_A_HIDDEN (0x02)`
: Hidden file

`_A_SYSTEM (0x04)`
: System file

`_A_VOLID (0x08)`
: Volume ID file

`_A_SUBDIR (0x10)`
: Subdirectory

`_A_ARCH (0x20)`
: Archive file

The results are returned in a `struct find_t` defined on `<dos.h>` as follows:

```
struct find_t {
char reserved[21];
unsigned char attrib;
unsigned short wr_time;
unsigned short wr_date;
unsigned long size;
char name[256];
};
```

See _dos_findnext.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See findfirst, and See findnext. Also see See opendir, and See readdir, which are Posix-standard.

Return Value

Zero if a match is found, DOS error code if not found (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <dos.h>

struct find_t f;

if ( !_dos_findfirst("*.DAT", &f, _A_ARCH | _A_RDONLY) )
{

do
{
printf("%-14s %10u %02u:%02u:%02u %02u/%02u/%04u\n",
f.name,
f.size,
```

```
(f.wr_time >> 11) & 0x1f,
(f.wr_time >> 5) & 0x3f,
(f.wr_time & 0x1f) * 2,
(f.wr_date >> 5) & 0x0f,
(f.wr_date & 0x1f),
((f.wr_date >> 9) & 0x7f) + 1980);
} while( !_dos_findnext(&f) );
}
```

_dos_findnext

Syntax

```
#include <dos.h>

unsigned int _dos_findnext(struct find_t *result);
```

Description

This finds the next file in the search started by `_dos_findfirst`. See See _dos_findfirst, for the description of `struct find_t`.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See findfirst, and See findnext. Also see See opendir, and See readdir, which are Posix-standard.

Return Value

Zero if a match is found, DOS error code if not found (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dos_getdate

Syntax

```
#include <dos.h>

void _dos_getdate(struct dosdate_t *date);
```

Description

This function gets the current date and fills the date structure with these values.

```
struct dosdate_t {
unsigned char day; /* 1-31 */
unsigned char month; /* 1-12 */
unsigned short year; /* 1980-2099 */
unsigned char dayofweek; /* 0-6, 0=Sunday */
};
```

See _dos_setdate. See _dos_gettime. See _dos_settime.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct dosdate_t date;

_dos_getdate(&date);
```

_dos_getdiskfree

Syntax

```
#include <dos.h>

unsigned int _dos_getdiskfree(unsigned int drive,
struct diskfree_t *diskspace);
```

Description

This function determines the free space on drive drive (0=default, 1=A:, 2=B:, etc.) and fills diskspace structure. The members of `struct diskfree_t` are defined by `<dos.h>` as follows:

```
struct diskfree_t {
unsigned short total_clusters;
unsigned short avail_clusters;
unsigned short sectors_per_cluster;
unsigned short bytes_per_sector;
};
```

Return Value

Returns with 0 if successful, non-zero on error (and sets `errno` to `EINVAL`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct diskfree_t df;
unsigned long freebytes;

if ( !_dos_getdiskfree(0, &df) )
{

freebytes = (unsigned long)df.avail_clusters *
(unsigned long)df.bytes_per_sector *
(unsigned long)df.sectors_per_cluster;
printf("There is %lu free bytes on the current drive.\n", freebytes);
}

else
printf("Unable to get free disk space.\n");
```

_dos_getdrive

Syntax

```
#include <dos.h>

void _dos_getdrive(unsigned int *p_drive);
```

Description

This function determine the current default drive and writes this value into p_drive (1=A:, 2=B:, etc.).

See _dos_setdrive.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned int drive;

_dos_getdrive(&drive);
```

```
printf("The current drive is %c:.\n", 'A' - 1 + drive);
```

_dos_getfileattr

Syntax

```
#include <dos.h>

unsigned int _dos_getfileattr(const char *filename,
unsigned int *p_attr);
```

Description

This function determines the attributes of given file and fills attr with it. Use the following constans (in DOS.H) to check this value.

`_A_NORMAL (0x00)`
: Normal file (no read/write restrictions)

`_A_RDONLY (0x01)`
: Read only file

`_A_HIDDEN (0x02)`
: Hidden file

`_A_SYSTEM (0x04)`
: System file

`_A_VOLID (0x08)`
: Volume ID file

`_A_SUBDIR (0x10)`
: Subdirectory

`_A_ARCH (0x20)`
: Archive file

See _dos_setfileattr.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See _chmod. Also see See chmod, See access, and See stat, which are Posix-standard.

Return Value

Returns with 0 if successful and DOS error value on error (and sets `errno`=ENOENT).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned int attr;

if ( !_dos_getfileattr("FOO.DAT", &attr) )
{

puts("FOO.DAT attributes are:");
if ( attr & _A_ARCH ) puts("Archive");
if ( attr & _A_RDONLY ) puts("Read only");
if ( attr & _A_HIDDEN ) puts("Hidden");
if ( attr & _A_SYSTEM ) puts("Is it part of DOS ?");
if ( attr & _A_VOLID ) puts("Volume ID");
if ( attr & _A_SUBDIR ) puts("Directory");
}

else
puts("Unable to get FOO.DAT attributes.");
```

_dos_getftime

Syntax

```
#include <dos.h>

unsigned int _dos_getftime(int handle,
unsigned int *p_date, unsigned *p_time);
```

Description

This function gets the date and time of the given file and puts these values into p_date and p_time variable. The meaning of DOS date in the p_date variable is the following:

```
F E D C B A 9 8 7 6 5 4 3 2 1 0 (bits)
X X X X X X X X X X X X X X X X
*-----------------------* *-----------* *---------------*
year month day

year = 0-119 (relative to 1980)
month = 1-12
day = 1-31
```

The meaning of DOS time in the p_time variable is the following:

```
F E D C B A 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
*---------------* *-------------------* *---------------*
hours minutes seconds

hours = 0-23
minutes = 0-59
seconds = 0-29 in two-second intervals
```

See _dos_setftime.

This function cannot be used to return last access and creation date and time, even on systems where the LFN API (See _use_lfn, LFN) is available. See See _lfn_get_ftime, for a function that can be used to get the other two times. Also see See fstat, which is Posix-standard.

Return Value

Returns 0 if successful and return DOS error on error (and sets `errno`=EBADF).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned int handle, date, time;

_dos_open("FOO.DAT", O_RDWR, &handle);
_dos_getftime(handle, &date, &time);
_dos_close(handle);
printf("FOO.DAT date and time is: %04u-%02u-%02u %02u:%02u:%02u.\n",
/* year month day */
((date >> 9) & 0x7F) + 1980U, (date >> 5) & 0x0F, date & 0x1F,
/* hour minute second */
(time >> 11) & 0x1F, (time >> 5) & 0x3F, (time & 0x1F) * 2);
```

_dos_gettime

Syntax

```
#include <dos.h>

void _dos_gettime(struct dostime_t *time);
```

Description

This function gets the current time and fills the time structure with these values.

```
struct dostime_t {
unsigned char hour; /* 0-23 */
unsigned char minute; /* 0-59 */
unsigned char second; /* 0-59 */
unsigned char hsecond; /* 0-99 */
};
```

See _dos_settime. See _dos_getdate. See _dos_setdate.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct dostime_t time;

_dos_gettime(&time);
```

_dos_lk64

Syntax

```
#include <io.h>

int _dos_lk64(int fd, long long offset, long long length);
```

Description

Adds an advisory lock to the specified region of the file.

Arguments offset and length must be of type `long long`, thus enabling you to lock with offsets and lengths as large as ~2^63 (FAT16 limits this to ~2^31; FAT32 limits this to 2^32-2).

See _dos_unlk64.

Return Value

Zero if the lock was added, nonzero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dos_lock

Syntax

```
#include <io.h>

int _dos_lock(int fd, long offset, long length);
```

Description

Adds an advisory lock to the specified region of the file.

Return Value

Zero if the lock was added, nonzero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dos_open

Syntax

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

unsigned int _dos_open(const char *filename, unsigned short mode,
int *handle);
```

Description

This is a direct connection to the MS-DOS open function call (%ah = 0x3D). This function opens the given file with the given mode and puts handle of file into handle if openning is successful. Meaning of mode parameter is the following:

Access mode bits (in FCNTL.H):

`O_RDONLY (_O_RDONLY) 0x00`
: Open for read only

`O_WRONLY (_O_WRONLY) 0x01`
: Open for write only

`O_RDWR (_O_RDWR) 0x02`
: Open for read and write

Sharing mode bits (in SHARE.H):

`SH_COMPAT (_SH_COMPAT) 0x00`
: Compatibility mode

`SH_DENYRW (_SH_DENYRW) 0x10`
: Deny read/write mode

`SH_DENYWR (_SH_DENYWR) 0x20`
: Deny write mode

`SH_DENYRD (_SH_DENYRD) 0x30`
: Deny read mode

`SH_DENYNO (_SH_DENYNO) 0x40`
: Deny none mode

Inheritance bits (in FCNTL.H):

`O_NOINHERIT (_O_NOINHERIT) 0x80`
: File is not inherited by child process

See also See _dos_creat, See _dos_creatnew, See _dos_read, See _dos_write, and See _dos_close.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See _open, See _creat, and See _creatnew. Also see See open, and See creat, which are Posix-standard.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno` to EACCES, EINVAL, EMFILE or ENOENT).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;

if ( !_dos_open("FOO.DAT", O_RDWR, &handle) )
puts("Wow, file opening was successful !");
```

_dos_read

Syntax

```
#include <dos.h>

unsigned int _dos_read(int handle, void *buffer, unsigned int count,
unsigned int *result);
```

Description

This is a direct connection to the MS-DOS read function call (%ah = 0x3F). No conversion is done on the data; it is read as raw binary data. This function reads from handle into buffer count bytes. count value may be arbitrary size (for example > 64KB). It puts number of bytes read into result if reading is successful.

See also See _dos_open, See _dos_creat, See _dos_creatnew, See _dos_write, and See _dos_close.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno` to EACCES or EBADF)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;
unsigned int result;
char *filebuffer;

if ( !_dos_open("FOO.DAT", O_RDONLY, &handle) )
{

puts("FOO.DAT openning was successful.");
if ( (filebuffer = malloc(130000)) != NULL )
{
if ( !_dos_read(handle, buffer, 130000, &result) )
printf("%u bytes read from FOO.DAT.\n", result);
else
puts("Reading error.");
...
/* Do something with filebuffer. */
...
}
_dos_close(handle);
}
```

_dos_setdate

Syntax

```
#include <dos.h>

unsigned int _dos_setdate(struct dosdate_t *date);
```

Description

This function sets the current date. The dosdate_t structure is as follows:

```
struct dosdate_t {
unsigned char day; /* 1-31 */
unsigned char month; /* 1-12 */
unsigned short year; /* 1980-2099 */
unsigned char dayofweek; /* 0-6, 0=Sunday */
};
```

dayofweek field has no effect at this function call.

See _dos_getdate. See _dos_gettime. See _dos_settime.

Return Value

Returns 0 if successful and non-zero on error (and sets `errno`=EINVAL).

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
struct dosdate_t date;

date->year = 1999;
date->month = 12;
date->day = 31;
if ( !_dos_setdate(&date) )
puts("It was a valid date.");
```

_dos_setdrive
Syntax
```
#include <dos.h>

void _dos_setdrive(unsigned int drive, unsigned int *p_drives);
```

Description
This function set the current default drive based on drive (1=A:, 2=B:, etc.) and determines the number of available logical drives and fills p_drives with it.

See _dos_getdrive.

Return Value
None.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
unsigned int available_drives;

/* The current drive will be A: */
_dos_setdrive(1, &available_drives);
printf("Number of available logical drives %u.\n", available_drives);
```

_dos_setfileattr
Syntax
```
#include <dos.h>

unsigned int _dos_setfileattr(const char *filename, unsigned int attr);
```

Description
This function sets the attributes of given file. Use the following constans in DOS.H to create attr parameter:

`_A_NORMAL (0x00)`
Normal file (no read/write restrictions)

`_A_RDONLY (0x01)`
Read only file

`_A_HIDDEN (0x02)`
Hidden file

`_A_SYSTEM (0x04)`
System file

`_A_VOLID (0x08)`
Volume ID file

`_A_SUBDIR (0x10)`
: Subdirectory

`_A_ARCH (0x20)`
: Archive file

See _dos_getfileattr.

This function does not support long filenames, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See _chmod. Also see See chmod, which is Posix-standard.

Return Value

Returns with 0 if successful and DOS error value on error (and sets `errno` to ENOENT or EACCES).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if ( !_dos_setfileattr("FOO.DAT", _A_RDONLY | _A_HIDDEN) )
puts("FOO.DAT is hidden now.");
```

_dos_setftime

Syntax

```
#include <dos.h>

unsigned int _dos_setftime(int handle,
unsigned int date, unsigned time);
```

Description

This function sets the date and time of the given file. The meaning of DOS date in the date variable is the following:

```
F E D C B A 9 8 7 6 5 4 3 2 1 0 (bits)
x x x x x x x x x x x x x x x x
*-----------* *-----* *-------*
year month day

year = 0-119 (relative to 1980)
month = 1-12
day = 1-31
```

The meaning of DOS time in the time variable is the following:

```
F E D C B A 9 8 7 6 5 4 3 2 1 0 (bits)
x x x x x x x x x x x x x x x x
*-------* *---------* *-------*
hours minutes seconds

hours = 0-23
minutes = 0-59
seconds = 0-29 in two-second intervals
```

See _dos_getftime.

This function cannot be used to set the last access date and time, even on systems where the LFN API (See _use_lfn, LFN) is available. For LFN-aware functions with similar functionality see See utime, which is Posix-standard, and see See utimes.

Return Value

Returns 0 if successful and return DOS error on error (and sets `errno`=EBADF).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct dosdate_t d;
struct dostime_t t;
unsigned int handle, date, time;

_dos_open("FOO.DAT", O_RDWR, &handle);
_dos_getdate(&d);
_dos_gettime(&t);
date = ((d.year - 1980) << 9) | (d.month << 5) | d.day;
time = (t.hour << 11) | (t.minute << 5) | (t.second / 2);
_dos_setftime(handle, date, time);
_dos_close(handle);
```

_dos_settime

Syntax

```
#include <dos.h>

void _dos_settime(struct dostime_t *time);
```

Description

This function sets the current time. The time structure is as follows:

```
struct dostime_t {
unsigned char hour; /* 0-23 */
unsigned char minute; /* 0-59 */
unsigned char second; /* 0-59 */
unsigned char hsecond; /* 0-99 */
};
```

See _dos_gettime. See _dos_getdate. See _dos_setdate.

Return Value

Returns 0 if successful and non-zero on error (and sets `errno`=EINVAL).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct dostime_t time;

time->hour = 23;
time->minute = 59;
time->second = 59;
time->hsecond = 99;
if ( !_dos_settime(&time) )
puts("It was a valid time.");
```

_dos_unlk64

Syntax

```
#include <io.h>

int _dos_unlk64(int fd, long long offset, long long length);
```

Description

Removes an advisory lock to the specified region of the file.

Arguments offset and length must be of type `long long`, thus enabling you to unlock with offsets and lengths as large as ~2^63 (FAT16 limits this to ~2^31; FAT32 limits this to 2^32-2).

See _dos_lk64.

Return Value

Zero if the lock was removed, nonzero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dos_unlock

Syntax

```
#include <io.h>

_dos_unlock(int fd, long offset, long length);
```

Description

Removes an advisory lock to the specified region of the file.

Return Value

Zero if the lock was removed, nonzero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dos_write

Syntax

```
#include <dos.h>

unsigned int _dos_write(int handle,
const void *buffer, unsigned int count,
unsigned int *result);
```

Description

This is a direct connection to the MS-DOS write function call (%ah = 0x40). No conversion is done on the data; it is written as raw binary data. This function writes count bytes from buffer to handle. count value may be arbitrary size (e.g. > 64KB). It puts the number of bytes written into result if writing is successful.

See also See _dos_open, See _dos_creat, See _dos_creatnew, See _dos_read, and See _dos_close.

Return Value

Returns 0 if successful or DOS error code on error (and sets `errno` to EACCES or EBADF)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int handle;
unsigned int result;
char *filebuffer;

if ( !_dos_creat("FOO.DAT", _A_ARCH, &handle) )
{

puts("FOO.DAT creating was successful.");
if ( (filebuffer = malloc(130000)) != NULL )
{
...
/* Put something into filebuffer. */
...
if ( !_dos_write(handle, buffer, 130000, &result) )
```

```
printf("%u bytes written into FOO.DAT.", result);
else
puts("Writing error.");
}
_dos_close(handle);
}
```

_doscan

Syntax

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

int _doscan(FILE *file, const char *format, va_list argp);
```

Description

This is an internal function that is used by all the `scanf` style functions, which simply pass their format, arguments, and stream to this function.

See scanf, for a discussion of the allowed formats and arguments.

Return Value

The number of characters successfully scanned is returned, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
TODO: This example is bogus now!
TODO: Rewrite this example!

int x, y;
int *args[2];
args[0] = &x;
args[1] = &y;
_doscan(stdin, "%d %d", args);
```

_doserrno

Syntax

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

Description

Whenever a DOS call returns a failure indication, this variable is assigned the value of the error code returned by the failed DOS call.

For a list of the error codes and their short descriptions, see See dosexterr.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_doserrno = 0;
fprintf (stdprn, "Hello, world!\r\n\f");
if (_doserrno == 0x1c)
fprintf (stderr, "The printer is out of paper!\n");
```

__dosexec_find_on_path

Syntax

```
#include <stdio.h>
#include <libc/dosexec.h>
```

```
char *__dosexec_find_on_path(const char *program,
char *envp[], char *buf);
```

Description

This function searches for a program using a given path. The program is searched for using a known set of executable extensions, e.g. `.exe`. These executable extensions are described for the `spawn*()` function (See spawn*).

Pass the program name in program, the environment array in envp and the output buffer in buf. envp is an array of pointers to the environment variables; it must be terminated with a `NULL` pointer. buf must be large enough to hold at least `FILENAME_MAX` bytes.

envp controls where `__dosexec_find_on_path` looks for the program. If envp is `NULL`, then only the current directory is searched. If envp contains the `PATH` environment variable whose value lists several directories, then these directories are also searched. The global variable `environ` is usually passed for envp.

Return Value

If the function finds the program, with or without one of the known executable extensions, either in the current directory or along the `PATH` as recorded in envp, it puts the full pathname into buf and returns a pointer to buf. Otherwise, it returns `NULL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char shellpath[FILENAME_MAX];
extern char * __dosexec_find_on_path (const char *, char *[], char *);
extern char **environ;

/* See if we can find "/bin/sh.exe", "/bin/sh.com", etc. */
if (__dosexec_find_on_path ("/bin/sh", (char **)0, shellpath)
/* If not found, look for "sh" along the PATH. */
|| __dosexec_find_on_path ("sh", environ, shellpath))
printf ("/bin/sh found as ‘%s’\n", shellpath);
```

dosexterr

Syntax

```
#include <dos.h>

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

Description

This function reads extended error information from DOS and fills p_error structure.

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

Values for DOS extended error code (exterr field):

```
--- DOS 2.0+ ---
00h (0) no error
01h (1) function number invalid
02h (2) file not found
03h (3) path not found
04h (4) too many open files (no handles available)
05h (5) access denied
06h (6) invalid handle
07h (7) memory control block destroyed
08h (8) insufficient memory
09h (9) memory block address invalid
0Ah (10) environment invalid (usually >32K in length)
```

0Bh (11) format invalid
0Ch (12) access code invalid
0Dh (13) data invalid
0Eh (14) reserved
0Eh (14) (PTS-DOS 6.51+, S/DOS 1.0+) fixup overflow
0Fh (15) invalid drive
10h (16) attempted to remove current directory
11h (17) not same device
12h (18) no more files
---DOS 3.0+ (INT 24 errors)---
13h (19) disk write-protected
14h (20) unknown unit
15h (21) drive not ready
16h (22) unknown command
17h (23) data error (CRC)
18h (24) bad request structure length
19h (25) seek error
1Ah (26) unknown media type (non-DOS disk)
1Bh (27) sector not found
1Ch (28) printer out of paper
1Dh (29) write fault
1Eh (30) read fault
1Fh (31) general failure
20h (32) sharing violation
21h (33) lock violation
22h (34) disk change invalid (ES:DI -> media ID structure)(see #01681)
23h (35) FCB unavailable
23h (35) (PTS-DOS 6.51+, S/DOS 1.0+) bad FAT
24h (36) sharing buffer overflow
25h (37) (DOS 4.0+) code page mismatch
26h (38) (DOS 4.0+) cannot complete file operation (EOF / out of input)
27h (39) (DOS 4.0+) insufficient disk space
28h-31h reserved
---OEM network errors (INT 24)---
32h (50) network request not supported
33h (51) remote computer not listening
34h (52) duplicate name on network
35h (53) network name not found
36h (54) network busy
37h (55) network device no longer exists
38h (56) network BIOS command limit exceeded
39h (57) network adapter hardware error
3Ah (58) incorrect response from network
3Bh (59) unexpected network error
3Ch (60) incompatible remote adapter
3Dh (61) print queue full
3Eh (62) queue not full
3Fh (63) not enough space to print file
40h (64) network name was deleted
41h (65) network: Access denied
(DOS 3.0+ [maybe 3.3+???]) codepage switching not possible
(see also INT 21/AX=6602h,INT 2F/AX=AD42h)
42h (66) network device type incorrect
43h (67) network name not found
44h (68) network name limit exceeded
45h (69) network BIOS session limit exceeded
46h (70) temporarily paused
47h (71) network request not accepted
48h (72) network print/disk redirection paused
49h (73) network software not installed
(LANtastic) invalid network version
4Ah (74) unexpected adapter close
(LANtastic) account expired
4Bh (75) (LANtastic) password expired
4Ch (76) (LANtastic) login attempt invalid at this time
4Dh (77) (LANtastic v3+) disk limit exceeded on network node
4Eh (78) (LANtastic v3+) not logged in to network node
4Fh (79) reserved

---end of errors reportable via INT 24---
50h (80) file exists
51h (81) (undoc) duplicated FCB
52h (82) cannot make directory
53h (83) fail on INT 24h
---network-related errors (non-INT 24)---
54h (84) (DOS 3.3+) too many redirections / out of structures
55h (85) (DOS 3.3+) duplicate redirection / already assigned
56h (86) (DOS 3.3+) invalid password
57h (87) (DOS 3.3+) invalid parameter
58h (88) (DOS 3.3+) network write fault
59h (89) (DOS 4.0+) function not supported on network / no process slots available
5Ah (90) (DOS 4.0+) required system component not installed / not frozen
5Bh (91) (DOS 4.0+,NetWare4) timer server table overflowed
5Ch (92) (DOS 4.0+,NetWare4) duplicate in timer service table
5Dh (93) (DOS 4.0+,NetWare4) no items to work on
5Fh (95) (DOS 4.0+,NetWare4) interrupted / invalid system call
64h (100) (MSCDEX) unknown error
64h (100) (DOS 4.0+,NetWare4) open semaphore limit exceeded
65h (101) (MSCDEX) not ready
65h (101) (DOS 4.0+,NetWare4) exclusive semaphore is already owned
66h (102) (MSCDEX) EMS memory no longer valid
66h (102) (DOS 4.0+,NetWare4) semaphore was set when close attempted
67h (103) (MSCDEX) not High Sierra or ISO-9660 format
67h (103) (DOS 4.0+,NetWare4) too many exclusive semaphore requests
68h (104) (MSCDEX) door open
68h (104) (DOS 4.0+,NetWare4) operation invalid from interrupt handler
69h (105) (DOS 4.0+,NetWare4) semaphore owner died
6Ah (106) (DOS 4.0+,NetWare4) semaphore limit exceeded
6Bh (107) (DOS 4.0+,NetWare4) insert drive B: disk into A: / disk changed
6Ch (108) (DOS 4.0+,NetWare4) drive locked by another process
6Dh (109) (DOS 4.0+,NetWare4) broken pipe
6Eh (110) (DOS 5.0+,NetWare4) pipe open/create failed
6Fh (111) (DOS 5.0+,NetWare4) pipe buffer overflowed
70h (112) (DOS 5.0+,NetWare4) disk full
71h (113) (DOS 5.0+,NetWare4) no more search handles
72h (114) (DOS 5.0+,NetWare4) invalid target handle for dup2
73h (115) (DOS 5.0+,NetWare4) bad user virtual address / protection violation
74h (116) (DOS 5.0+) VIOKBD request
74h (116) (NetWare4) error on console I/O
75h (117) (DOS 5.0+,NetWare4) unknown category code for IOCTL
76h (118) (DOS 5.0+,NetWare4) invalid value for verify flag
77h (119) (DOS 5.0+,NetWare4) level four driver not found by DOS IOCTL
78h (120) (DOS 5.0+,NetWare4) invalid / unimplemented function number
79h (121) (DOS 5.0+,NetWare4) semaphore timeout
7Ah (122) (DOS 5.0+,NetWare4) buffer too small to hold return data
7Bh (123) (DOS 5.0+,NetWare4) invalid character or bad file-system name
7Ch (124) (DOS 5.0+,NetWare4) unimplemented information level
7Dh (125) (DOS 5.0+,NetWare4) no volume label found
7Eh (126) (DOS 5.0+,NetWare4) module handle not found
7Fh (127) (DOS 5.0+,NetWare4) procedure address not found
80h (128) (DOS 5.0+,NetWare4) CWait found no children
81h (129) (DOS 5.0+,NetWare4) CWait children still running
82h (130) (DOS 5.0+,NetWare4) invalid operation for direct disk-access handle
83h (131) (DOS 5.0+,NetWare4) attempted seek to negative offset
84h (132) (DOS 5.0+,NetWare4) attempted to seek on device or pipe
---JOIN/SUBST errors---
85h (133) (DOS 5.0+,NetWare4) drive already has JOINed drives
86h (134) (DOS 5.0+,NetWare4) drive is already JOINed
87h (135) (DOS 5.0+,NetWare4) drive is already SUBSTed
88h (136) (DOS 5.0+,NetWare4) can not delete drive which is not JOINed
89h (137) (DOS 5.0+,NetWare4) can not delete drive which is not SUBSTed
8Ah (138) (DOS 5.0+,NetWare4) can not JOIN to a JOINed drive
8Bh (139) (DOS 5.0+,NetWare4) can not SUBST to a SUBSTed drive
8Ch (140) (DOS 5.0+,NetWare4) can not JOIN to a SUBSTed drive
8Dh (141) (DOS 5.0+,NetWare4) can not SUBST to a JOINed drive
8Eh (142) (DOS 5.0+,NetWare4) drive is busy

8Fh (143) (DOS 5.0+,NetWare4) can not JOIN/SUBST to same drive
90h (144) (DOS 5.0+,NetWare4) directory must not be root directory
91h (145) (DOS 5.0+,NetWare4) can only JOIN to empty directory
92h (146) (DOS 5.0+,NetWare4) path is already in use for SUBST
93h (147) (DOS 5.0+,NetWare4) path is already in use for JOIN
94h (148) (DOS 5.0+,NetWare4) path is in use by another process
95h (149) (DOS 5.0+,NetWare4) directory previously SUBSTituted
96h (150) (DOS 5.0+,NetWare4) system trace error
97h (151) (DOS 5.0+,NetWare4) invalid event count for DosMuxSemWait
98h (152) (DOS 5.0+,NetWare4) too many waiting on mutex
99h (153) (DOS 5.0+,NetWare4) invalid list format
9Ah (154) (DOS 5.0+,NetWare4) volume label too large
9Bh (155) (DOS 5.0+,NetWare4) unable to create another TCB
9Ch (156) (DOS 5.0+,NetWare4) signal refused
9Dh (157) (DOS 5.0+,NetWare4) segment discarded
9Eh (158) (DOS 5.0+,NetWare4) segment not locked
9Fh (159) (DOS 5.0+,NetWare4) invalid thread-ID address

A0h (160) (DOS 5.0+) bad arguments
A0h (160) (NetWare4) bad environment pointer
A1h (161) (DOS 5.0+,NetWare4) invalid pathname passed to EXEC
A2h (162) (DOS 5.0+,NetWare4) signal already pending
A3h (163) (DOS 5.0+) uncertain media
A3h (163) (NetWare4) ERROR_124 mapping
A4h (164) (DOS 5.0+) maximum number of threads reached
A4h (164) (NetWare4) no more process slots
A5h (165) (NetWare4) ERROR_124 mapping
B0h (176) (MS-DOS 7.0) volume is not locked
B1h (177) (MS-DOS 7.0) volume is locked in drive
B2h (178) (MS-DOS 7.0) volume is not removable
B4h (180) (MS-DOS 7.0) lock count has been exceeded
B4h (180) (NetWare4) invalid segment number
B5h (181) (MS-DOS 7.0) a valid eject request failed
B5h (181) (DOS 5.0-6.0,NetWare4) invalid call gate
B6h (182) (DOS 5.0+,NetWare4) invalid ordinal
B7h (183) (DOS 5.0+,NetWare4) shared segment already exists
B8h (184) (DOS 5.0+,NetWare4) no child process to wait for
B9h (185) (DOS 5.0+,NetWare4) NoWait specified and child still running
BAh (186) (DOS 5.0+,NetWare4) invalid flag number
BBh (187) (DOS 5.0+,NetWare4) semaphore does not exist
BCh (188) (DOS 5.0+,NetWare4) invalid starting code segment
BDh (189) (DOS 5.0+,NetWare4) invalid stack segment
BEh (190) (DOS 5.0+,NetWare4) invalid module type (DLL can not be used as application)
BFh (191) (DOS 5.0+,NetWare4) invalid EXE signature
C0h (192) (DOS 5.0+,NetWare4) EXE marked invalid
C1h (193) (DOS 5.0+,NetWare4) bad EXE format (e.g. DOS-mode program)
C2h (194) (DOS 5.0+,NetWare4) iterated data exceeds 64K
C3h (195) (DOS 5.0+,NetWare4) invalid minimum allocation size
C4h (196) (DOS 5.0+,NetWare4) dynamic link from invalid Ring
C5h (197) (DOS 5.0+,NetWare4) IOPL not enabled
C6h (198) (DOS 5.0+,NetWare4) invalid segment descriptor privilege level
C7h (199) (DOS 5.0+,NetWare4) automatic data segment exceeds 64K
C8h (200) (DOS 5.0+,NetWare4) Ring2 segment must be moveable
C9h (201) (DOS 5.0+,NetWare4) relocation chain exceeds segment limit
CAh (202) (DOS 5.0+,NetWare4) infinite loop in relocation chain
CBh (203) (NetWare4) environment variable not found
CCh (204) (NetWare4) not current country
CDh (205) (NetWare4) no signal sent
CEh (206) (NetWare4) file name not 8.3
CFh (207) (NetWare4) Ring2 stack in use
D0h (208) (NetWare4) meta expansion is too long
D1h (209) (NetWare4) invalid signal number
D2h (210) (NetWare4) inactive thread
D3h (211) (NetWare4) file system information not available
D4h (212) (NetWare4) locked error
D5h (213) (NetWare4) attempted to execute non-family API call in DOS mode
D6h (214) (NetWare4) too many modules

```
D7h (215) (NetWare4) nesting not allowed
E6h (230) (NetWare4) non-existent pipe, or bad operation
E7h (231) (NetWare4) pipe is busy
E8h (232) (NetWare4) no data available for nonblocking read
E9h (233) (NetWare4) pipe disconnected by server
EAh (234) (NetWare4) more data available
FFh (255) (NetWare4) invalid drive
```

Values for error class (class field):

```
01h (1) out of resource (storage space or I/O channels)
02h (2) temporary situation (file or record lock)
03h (3) authorization / permission problem (denied access)
04h (4) internal system error (system software bug)
05h (5) hardware failure
06h (6) system failure (configuration file missing or incorrect)
07h (7) application program error
08h (8) not found
09h (9) bad format
0Ah (10) locked
0Bh (11) media error
0Ch (12) already exists / collision with existing item
0Dh (13) unknown / other
0Eh (14) (undoc) cannot
0Fh (15) (undoc) time
```

Values for suggested action (action field):

```
01h retry
02h delayed retry (after pause)
03h prompt user to reenter input
04h abort after cleanup
05h immediate abort ("panic")
06h ignore
07h retry after user intervention
```

Values for error locus (locus field):

```
01h unknown or not appropriate
02h block device (disk error)
03h network related
04h serial device (timeout)
04h (PTS-DOS 6.51+ & S/DOS 1.0+) character device
05h memory related
```

Return Value

Returns with the extended error code.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{

FILE *fp;
struct DOSERROR de;

fp = fopen("EXAMPLE.DAT","r");
if ( fp == NULL )
{
puts("Unable to open file for reading.");
dosexterr(&de);
printf("Extended DOS error information:\n");
printf("Extended error: %i\n",de.exterror);
printf("Class: %x\n",de.class);
printf("Action: %x\n",de.action);
printf("Error Locus: %x\n",de.locus);
}
```

```
return 0;
}
```

dosmemget

Syntax

```
#include <sys/movedata.h>

void dosmemget(int offset, int length, void *buffer);
```

Description

This function transfers data from MS-DOS's conventional memory space to the program's virtual address space. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The length is the number of bytes to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will go.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short shift_state;
dosmemget(0x417, 2, &shift_state);
if (shift_state & 0x0004)
/* Ctrl key pressed */;
```

dosmemgetb

Syntax

```
#include <sys/movedata.h>

void _dosmemgetb(unsigned long offset, size_t xfers, void *buffer);
```

Description

This function transfers data from MS-DOS's conventional memory space to the program's virtual address space, using only byte transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The xfers is the number of bytes to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will go.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short shift_state;
_dosmemgetb(0x417, 2, &shift_state);
if (shift_state & 0x0004)
/* Ctrl key pressed */;
```

dosmemgetl

Syntax

```
#include <sys/movedata.h>

void _dosmemgetl(unsigned long offset, size_t xfers, void *buffer);
```

Description

This function transfers data from MS-DOS's conventional memory space to the program's virtual address space, using only long-word (32-bit) transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The count is the number of long-words to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will go.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned long shift_state;
_dosmemgetl(0x417, 1, &shift_state);
if (shift_state & 0x0004)
/* Ctrl key pressed */;
```

dosmemgetw

Syntax

```
#include <sys/movedata.h>

void _dosmemgetw(unsigned long offset, size_t xfers, void *buffer);
```

Description

This function transfers data from MS-DOS's conventional memory space to the program's virtual address space, using only short-word (16-bit) transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The xfers is the number of words to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will go.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short shift_state;
_dosmemgetw(0x417, 1, &shift_state);
if (shift_state & 0x0004)
/* Ctrl key pressed */;
```

dosmemput

Syntax

```
#include <sys/movedata.h>

void dosmemput(const void *buffer, int length, int offset);
```

Description

This function transfers data from the program's virtual address space to MS-DOS's conventional memory space. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The length is the number of bytes to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will come from.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short save_screen[25][80];
dosmemput(save_screen, 80*2*25, 0xb8000);
```

dosmemputb

Syntax

```
#include <sys/movedata.h>

void _dosmemputb(const void *buffer, size_t xfers,
unsigned long offset);
```

Description

This function transfers data from the program's virtual address space to MS-DOS's conventional memory space, using only byte (8-bit) transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The xfers is the number of bytes to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will come from.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short save_screen[25][80];
_dosmemputb(save_screen, 0xb8000, 80*2*25);
```

dosmemputl

Syntax

```
#include <sys/movedata.h>

void _dosmemputl(const void *buffer, size_t xfers,
unsigned long offset);
```

Description

This function transfers data from the program's virtual address space to MS-DOS's conventional memory space, using only long-word (32-bit) transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The xfers is the number of long-words to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will come from.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short save_screen[25][80];
_dosmemputl(save_screen, 40*25, 0xb8000);
```

dosmemputw

Syntax

```
#include <sys/movedata.h>

void _dosmemputw(const void *buffer, size_t xfers,
unsigned long offset);
```

Description

This function transfers data from the program's virtual address space to MS-DOS's conventional memory space, using only short-word (16-bit) transfers. The offset is a physical address, which can be computed from a real-mode segment/offset pair as follows:

```
offset = segment * 16 + offset;
```

The xfers is the number of short-words to transfer, and buffer is a pointer to somewhere in your virtual address space (such as memory obtained from `malloc`) where the data will come from.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short save_screen[25][80];
_dosmemputw(save_screen, 0xb8000, 80*25);
```

dostrerr

Syntax

```
#include <dos.h>

int dostrerr(struct DOSERROR *p_error, struct DOSERROR_str *p_str);
```

Description

This function accepts the extended error structure from DOS (e.g., from the returned parameter from function `dosexterr`, See dosexterr) and returns the strings which describes that error structure in the structure pointed to by the second parameter. This function is a DOS analogue of the ANSI function `strerror` (See strerror), and can be used to print descriptive messages corresponding to the errors described in the DOSERROR structure.

For a list of the strings returned for each error number and type, see See dosexterr.

p_error must point to the following structure:

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

p_str must point to the following structure:

```
struct DOSERROR_STR {
char *exterror_str;
char *class_str;
char *action_str;
char *locus_str;
};
```

Return Value

If either pointer parameter is NULL, returns -1 and sets errno to EINVAL. If both parameters are not NULL, checks the value of each member of the DOSERROR parameter p_error. If each value is within the limits of valid error codes for that member, sets parameter p_str member fields with the corresponding string describing the error code. If any error code is outside of the valid values for that code, sets the corresponding p_str member to the string ‘‘Unknown error: ’’ followed by the decimal numeric value of the error code.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{

FILE *fp;
struct DOSERROR de;
struct DOSERROR_STR se;

fp = fopen("EXAMPLE.DAT","r");
if ( fp == NULL )
{
puts("Unable to open file for reading.");
dosexterr(&de);
dostrerr(&de, &se);
printf("Extended DOS error information:\n");
printf("Extended error: %i : %s\n",de.exterror,se.exterror_str);
printf("Class: %x : %s\n",de.class,se.class_str);
printf("Action: %x : %s\n",de.action,se.action_str);
printf("Error Locus: %x : %s\n",de.locus,se.locus_str);
}
return 0;
}
```

DPMI Overview

```
extern unsigned short __dpmi_error;
```

For most functions, the error returned from the DPMI server is stored in this variable.

```
typedef struct {
unsigned short offset16;
unsigned short segment;
} __dpmi_raddr;
```

This structure is used to hold a real-mode address, which consists of a segment:offset pair.

```
typedef struct {
unsigned long offset32;
unsigned short selector;
} __dpmi_paddr;
```

This structure is used to hold a protected-mode address, which consists of a selector:offset pair.

```
typedef struct {
unsigned long handle; /* 0, 2 */
unsigned long size; /* or count */ /* 4, 6 */
unsigned long address; /* 8, 10 */
} __dpmi_meminfo;
```

This structure is used by many functions that need to refer to blocks of 32-bit memory. The `size` field doubles as a count for those operations that want a count of something, or return a count.

```
typedef union {
struct {
unsigned long edi;
unsigned long esi;
unsigned long ebp;
unsigned long res;
unsigned long ebx;
unsigned long edx;
unsigned long ecx;
unsigned long eax;
} d;
struct {
unsigned short di, di_hi;
unsigned short si, si_hi;
unsigned short bp, bp_hi;
unsigned short res, res_hi;
unsigned short bx, bx_hi;
unsigned short dx, dx_hi;
unsigned short cx, cx_hi;
unsigned short ax, ax_hi;
unsigned short flags;
unsigned short es;
unsigned short ds;
unsigned short fs;
unsigned short gs;
unsigned short ip;
unsigned short cs;
unsigned short sp;
unsigned short ss;
} x;
struct {
unsigned char edi[4];
unsigned char esi[4];
unsigned char ebp[4];
unsigned char res[4];
unsigned char bl, bh, ebx_b2, ebx_b3;
unsigned char dl, dh, edx_b2, edx_b3;
unsigned char cl, ch, ecx_b2, ecx_b3;
unsigned char al, ah, eax_b2, eax_b3;
} h;
} __dpmi_regs;
```

This structure is used by functions that pass register information, such as simulating real-mode calls.

```
typedef struct {
```

```
unsigned char major;
unsigned char minor;
unsigned short flags;
unsigned char cpu;
unsigned char master_pic;
unsigned char slave_pic;
} __dpmi_version_ret;
```

This structure is used to return version information to the program.

```
typedef struct {
unsigned long largest_available_free_block_in_bytes;
unsigned long maximum_unlocked_page_allocation_in_pages;
unsigned long maximum_locked_page_allocation_in_pages;
unsigned long linear_address_space_size_in_pages;
unsigned long total_number_of_unlocked_pages;
unsigned long total_number_of_free_pages;
unsigned long total_number_of_physical_pages;
unsigned long free_linear_address_space_in_pages;
unsigned long size_of_paging_file_partition_in_pages;
unsigned long reserved[3];
} __dpmi_free_mem_info;
```

This structure is used to return information about the state of virtual memory in the system.

```
typedef struct {
unsigned long total_allocated_bytes_of_physical_memory_host;
unsigned long total_allocated_bytes_of_virtual_memory_host;
unsigned long total_available_bytes_of_virtual_memory_host;
unsigned long total_allocated_bytes_of_virtual_memory_vcpu;
unsigned long total_available_bytes_of_virtual_memory_vcpu;
unsigned long total_allocated_bytes_of_virtual_memory_client;
unsigned long total_available_bytes_of_virtual_memory_client;
unsigned long total_locked_bytes_of_memory_client;
unsigned long max_locked_bytes_of_memory_client;
unsigned long highest_linear_address_available_to_client;
unsigned long size_in_bytes_of_largest_free_memory_block;
unsigned long size_of_minimum_allocation_unit_in_bytes;
unsigned long size_of_allocation_alignment_unit_in_bytes;
unsigned long reserved[19];
} __dpmi_memory_info;
```

This is also used to return memory information, but by a different function.

```
typedef struct {
unsigned long data16[2];
unsigned long code16[2];
unsigned short ip;
unsigned short reserved;
unsigned long data32[2];
unsigned long code32[2];
unsigned long eip;
} __dpmi_callback_info;
```

This structure is used to install TSR programs.

```
typedef struct {
unsigned long size_requested;
unsigned long size;
unsigned long handle;
unsigned long address;
unsigned long name_offset;
unsigned short name_selector;
unsigned short reserved1;
unsigned long reserved2;
} __dpmi_shminfo;
```

This structure is used to manipulate shared memory regions.

DPMI Specification

To obtain the DPMI specification, Contact Intel and order document number 240977-001. Also, try ftp.qdeck.com:/pub/memory/dpmi* and http://www.delorie.com/djgpp/doc/dpmi/.

__dpmi_allocate_dos_memory

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_dos_memory(int _paragraphs,
int *_ret_selector_or_max);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0100

This function allocates DOS memory. You pass it the number of paragraphs ((bytes+15)>>4) to allocate. If it succeeds, it returns a segment (dos-style) and fills in _ret_selector_or_max with a selector (protected-mode) that you can use to reference the same memory. Note that it's the selector you use to free the block, not the segment.

Return Value

-1 on error, else the segment [0000..FFFF].

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_allocate_ldt_descriptors

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_ldt_descriptors(int count);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0000

Allocates count descriptors.

Return Value

-1 on error, else the first descriptor. Use `__dpmi_get_selector_increment_value` (See __dpmi_get_selector_increment_value) to figure out the remaining selectors.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
short sel = __dpmi_allocate_ldt_descriptors(1);
```

__dpmi_allocate_linear_memory

Syntax

```
#include <dpmi.h>
```

```
int __dpmi_allocate_linear_memory(__dpmi_meminfo *info, int commit);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0504 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This allocates a block of page-aligned linear address space. Pass a desired address (or zero for any) and a size. commit is 1 for committed pages, else they are uncommitted. It returns a handle and the actual address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_allocate_memory

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_memory(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0501

This allocates virtual memory. Fill in size, returns handle and address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_allocate_real_mode_callback

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_real_mode_callback(void (*_handler)(void),
__dpmi_regs *_regs,
__dpmi_raddr *_ret);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0303

This function gives you a real-mode address to pass to TSRs that gets reflected to your protected-mode handler. You pass it a register block to use; it gets filled in with the real-mode registers when your handler is called, and the registers are set from it when the handler returns.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_allocate_shared_memory

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_shared_memory(__dpmi_shminfo *info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0d00 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function allocates a shared memory block that can be accessed from different virtual machines. Fill the required length in `info->size_requested`. The function fills the rest of the structure: allocated length in `info->size`, block handle in `info->handle`, linear address in `info->address`, and the selector:offset of an ASCIIZ block name (up to 128 bytes long) in `info->name_selector` and `info->name_offset`, respectively.

The access to the shared memory block can be serialized by calling the `__dpmi_serialize_on_shared_memory` function (See __dpmi_serialize_on_shared_memory).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_allocate_specific_ldt_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_allocate_specific_ldt_descriptor(int _selector);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000d

This allocates the specific selector given.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_clear_debug_watchpoint

Syntax

```
#include <dpmi.h>

int __dpmi_clear_debug_watchpoint(unsigned long _handle);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0b01

Clear a debug watchpoint.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_create_alias_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_create_alias_descriptor(int _selector);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000a

Create a new selector with the same parameters as the given one.

Return Value

-1 on error, else the new selector.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_discard_page_contents

Syntax

```
#include <dpmi.h>

int __dpmi_discard_page_contents(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0703

Advises the server that the given pages are no longer needed and may be reclaimed. Fill in address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_dos_memory

Syntax

```
#include <dpmi.h>
```

```
int __dpmi_free_dos_memory(int _selector);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0101

This function frees the dos memory allocated by See __dpmi_allocate_dos_memory. Remember to pass the selector and not the segment.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_ldt_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_free_ldt_descriptor(int descriptor);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0001

This function frees a single descriptor, even if it was allocated as one of many.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
__dpmi_free_ldt_descriptor(sel);
```

__dpmi_free_memory

Syntax

```
#include <dpmi.h>

int __dpmi_free_memory(unsigned long _handle);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0502

This frees a block of virtual memory.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_physical_address_mapping

Syntax

```
#include <dpmi.h>

int __dpmi_free_physical_address_mapping(__dpmi_meminfo *info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0801 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function unmaps a physical device mapped with See __dpmi_physical_address_mapping. Fill in the linear address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_real_mode_callback

Syntax

```
#include <dpmi.h>

int __dpmi_free_real_mode_callback(__dpmi_raddr *_addr);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0303

This function frees the real-mode callback address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_serialization_on_shared_memory

Syntax

```
#include <dpmi.h>

int __dpmi_free_serialization_on_shared_memory(unsigned long handle,
int flags);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0d03 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function frees the serialization on shared memory block specified by its handle handle. The bit-mapped variable flags defines the following bits:

bit 0
: If set, release shared serialization (as opposed to exclusive serialization).

bit 1
: If set, free pending serialization.

bits 2-15
: Reserved (should be zero).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_free_shared_memory

Syntax

```
#include <dpmi.h>

int __dpmi_free_shared_memory(unsigned long handle);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0d01 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function frees the shared memory block specified by the given handle. The handle becomes invalid after this call.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_and_disable_virtual_interrupt_state

Syntax

```
#include <dpmi.h>

int __dpmi_get_and_disable_virtual_interrupt_state(void);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0900

This function disables interrupts, and returns the previous setting.

Return Value

The previous setting.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_and_enable_virtual_interrupt_state

Syntax

```
#include <dpmi.h>

int __dpmi_get_and_enable_virtual_interrupt_state(void);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0901

This function enables interrupts, and returns the previous setting.

Return Value

The previous setting.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_and_set_virtual_interrupt_state

Syntax

```
#include <dpmi.h>

int __dpmi_get_and_set_virtual_interrupt_state(int _old_state);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AH = 0x09

This function restores the interrupt state from a previous call to `__dpmi_get_and_disable_virtual_interrupt_state` (See __dpmi_get_and_disable_virtual_interrupt_state) or `__dpmi_get_and_enable_virtual_interrupt_state` (See __dpmi_get_and_enable_virtual_interrupt_state).

Return Value

The previous setting.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_capabilities

Syntax

```
#include <dpmi.h>

int __dpmi_get_capabilities(int *flags, char *vendor_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0401 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

Gets the capabilities of the server. flags are as follows:

```
---- ---X = 1="page accessed/dirty" supported
---- --X- = 1="exceptions restartble" supported
---- -X-- = 1="device mapping" supported
---- X--- = 1="map conventional memory" supported
---X ---- = 1="demand zero-fill" supported
--X- ---- = 1="write-protect client" supported
-X-- ---- = 1="write-protect host" supported
```

The vendor info is a 128-byte buffer:

```
[0] host major number
[1] host minor number
[2..127] vendor name
```

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_coprocessor_status

Syntax

```
#include <dpmi.h>

int __dpmi_get_coprocessor_status(void);
```

Description

Please refer to See DPMI Specification, for details on DPMI function call operation. Also see See DPMI Overview, for general information.

DPMI function AX = 0x0e00 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

Return Value

-1 on error, else returns the processor status flags. Here's the meaning of each set bit:

bit 0
: If set, co-processor is enabled. If reset, co-processor is disabled.

bit 1
: If set, the application is emulating the co-processor.

bit 2
: If set, the numeric co-processor is present.

bit 3
: If set, the DPMI host is emulating the co-processor.

bits 4-7
: The co-processor type:

  `0000`
  : none

  `0010`
  : 80287

  `0011`
  : 80387

  `0100`
  : 80486 with a numeric processor

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_get_descriptor(int _selector, void *_buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000b

This function fills the 8-byte buffer pointed to by _buffer with the parameters of the descriptor whose selector is passed in _selector. The data has the following format:

```
[0] XXXX XXXX = segment limit [7:0]
[1] XXXX XXXX = segment limit [15:8]
[2] XXXX XXXX = base address [7:0]
[3] XXXX XXXX = base address [15:8]
[4] XXXX XXXX = base address [23:16]
[5] ---- XXXX = type; see details below
[5] ---X ---- = 0=system, 1=application (must be 1)
[5] -XX- ---- = privilege level, usually 3 (binary 11)
[5] X--- ---- = 0=absent, 1=present; usually 1
[6] ---- XXXX = segment limit [19:16]
[6] ---X ---- = available for user; see details below
[6] --0- ---- = must be zero
[6] -X-- ---- = 0=16-bit 1=32-bit; usually 1
[6] X--- ---- = 0=byte-granular (small) 1=page-granular (big)
[7] XXXX XXXX = base address [31:24]
```

Here's an alternative view of the layout that treats the buffer as an array of 4 16-bit words (i.e., `unsigned shorts`):

```
[0] XXXX XXXX XXXX XXXX = segment limit [15:0]
[1] XXXX XXXX XXXX XXXX = base address [15:0]
[2] ---- ---- XXXX XXXX = base address [23:16]
[2] ---- XXXX ---- ---- = type; see details below
[2] ---1 ---- ---- ---- = 0=system, 1=application; must be 1
[2] -XX- ---- ---- ---- = privilege level, usually 3 (binary 11)
[2] X--- ---- ---- ---- = 0=absent, 1=present; usually 1
[3] ---- ---- ---- XXXX = segment limit [19:16]
[3] ---- ---- ---X ---- = available for user; see details below
[3] ---- ---- --0- ---- = must be zero
[3] ---- ---- -X-- ---- = 0=16-bit 1=32-bit; usually 1
[3] ---- ---- X--- ---- = 0=byte-granular (small) 1=page-granular (big)
[3] XXXX XXXX ---- ---- = base address [31:24]
```

Special considerations apply to some of the fields:

Segment Limit fields
: The segment limit is specified as a 20-bit number. This number is interpreted as a number of bytes if the granularity bit (bit 7 of byte 6) is not set, and as a number of 4KB pages if the granularity bit is set. Offsets larger than the limit will generate a GPF, the General Protection Fault exception.

  For expand-down data segments (see below), the segment limit is the *lower* limit of the segment; the upper limit is either 0xffffffff or 0xffff, depending on whether the size bit is set (32-bit default size) or not (16-bit default size). For expand-down segments, values of offset *less* than the segment limit result in a GPF.

Base Address fields
: Segment base address should generally be 16-byte aligned. This is not required, but it maximizes performance by aligning code and data on 16-byte boundaries.

Type field
: This field has different meanings depending on whether the descriptor is for code or data segment. For code

segments, the meaning is as follows:

```
---X = 0=not accessed, 1=accessed
--1- = 0=execute only, 1=execute/read; must be 1
-0-- = 0=non-conforming, 1=conforming; must be 0
1--- = 0=data segment, 1=code segment
```

The accessed/not accessed bit indicates whether the segment has been accessed since the last time the bit was cleared. This bit is set whenever the segment selector is loaded into a segment register, and the bit then remains set until explicitly cleared. This bit can be used for debugging purposes.

The read bit must be set to allow reading data from the code segment, which is done in several cases by the library. The DPMI spec (See DPMI Specification) requires this bit to be 1 for code segments.

The conforming bit must be cleared so that transfer of execution into this segment from a less-privileged segment will result in a GPF. The DPMI spec (See DPMI Specification) requires this bit to be 0 for code segments.

For data segments, the meaning of the `type` field is as follows:

```
---X = 0=not accessed, 1=accessed
--X- = 0=read-only, 1=read/write
-X-- = 0=expand-up, 1=expand-down; usually 0
0--- = 0=data segment, 1=code segment
```

The accessed/not accessed bit has the same meaning as for code segments. The expand up/down bit is meant to be 1 for stack segments whose size should be changed dynamically, whereby changing the limit adds the additional space to the bottom of the stack; for data segments and statically-sized stack segments, this bit is usually zero.

Present bit
: If this bit is clear, a segment-not-present exception will be generated when the selector is loaded into a segment register, and all the fields of the descriptor except the privilege level and the system/application bit are available for CPU/OS to store their own data. Don't clear this bit unless you know what you are doing.

Available bit
: This bit is left for the application's use. It is neither set nor cleared by the DPMI server.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_descriptor_access_rights

Syntax

```
#include <dpmi.h>

int __dpmi_get_descriptor_access_rights(int _selector);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

This function returns the access rights byte from the `lar` opcode.

Return Value

The access byte. See __dpmi_set_descriptor_access_rights, for the details about the access information returned. Also see See __dpmi_get_descriptor.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_extended_exception_handler_vector_pm

Syntax

```
#include <dpmi.h>

int __dpmi_get_extended_exception_handler_vector_pm(
int vector, __dpmi_paddr *address
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0210 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This gets the function that handles protected mode exceptions.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_extended_exception_handler_vector_rm

Syntax

```
#include <dpmi.h>

int __dpmi_get_extended_exception_handler_vector_rm(
int vector, __dpmi_paddr *address
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0211 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function gets the handler for real-mode exceptions.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_free_memory_information

Syntax

```
#include <dpmi.h>

int __dpmi_get_free_memory_information(__dpmi_free_mem_info *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0500

This function returns information about available memory. Unsupported fields will have -1 (0xfffffff) in them.

Return Value

Zero. This always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_memory_block_size_and_base

Syntax

```
#include <dpmi.h>

int __dpmi_get_memory_block_size_and_base(__dpmi_meminfo *info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x050a (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

Pass the handle. It fills in the address and size.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_memory_information

Syntax

```
#include <dpmi.h>

int __dpmi_get_memory_information(__dpmi_memory_info *buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x050b (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function returns virtual memory information, as follows:

`total_allocated_bytes_of_physical_memory_host`
: The total amount of allocated physical memory controlled by the DPMI host.

`total_allocated_bytes_of_virtual_memory_host`
: The total amount of allocated virtual memory controlled by the DPMI host.

`total_available_bytes_of_virtual_memory_host`
: The total amount of available virtual memory controlled by the DPMI host.

`total_allocated_bytes_of_virtual_memory_vcpu`
: The amount of virtual memory allocated by the DPMI host for the current virtual machine.

`total_available_bytes_of_virtual_memory_vcpu`
: The amount of virtual memory available for the current virtual machine.

`total_allocated_bytes_of_virtual_memory_client`
: The amount of virtual memory allocated by the DPMI host for the current client (that is, for the calling program).

`total_available_bytes_of_virtual_memory_client`

The amount of virtual memory available to the current client.

`total_locked_bytes_of_memory_client`
The amount of memory locked by the calling program.

`max_locked_bytes_of_memory_client`
Maximum locked memory for the current client.

`highest_linear_address_available_to_client`
The highest linear address available to the calling program.

`size_in_bytes_of_largest_free_memory_block`
Size of the largest available memory block.

`size_of_minimum_allocation_unit_in_bytes`
Size of the smallest block that can be allocated.

`size_of_allocation_alignment_unit_in_bytes`
The alignment of allocated memory blocks.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_multiple_descriptors

Syntax

```
#include <dpmi.h>

int __dpmi_get_multiple_descriptors(int count, void *buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000e (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function gets a list of selectors' parameters. The buffer pointed to by buffer must be prefilled with selector values, and will contain the parameters on return:

```
[0x00:2] selector #1 (pass)
[0x02:8] parameters #1 (returned)
[0x0a:2] selector #2 (pass)
[0x0c:8] parameters #2 (returned)
...
```

Return Value

Returns count if successful, the negative of the number of descriptors copied if failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_page_attributes

Syntax

```
#include <dpmi.h>

int __dpmi_get_page_attributes(__dpmi_meminfo *info, short *buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0506 (DPMI 1.0 only). Supported by CWSDPMI, but not by Windows.

This function retrieves the attributes of a number of pages. Pass the handle in `info->handle`, offset of first page (relative to start of block) in `info->address`, and number of pages in `info->count`. The buffer buffer gets filled in with the attributes. For each page, a 16-bit attribute word in buffer defines the attributes of that page as follows:

bits 0-2
: Page type:

  `000`
  : uncommitted

  `001`
  : committed

  `010`
  : mapped

bit 3
: If set, the page is read/write. If cleared, the page is read-only.

bit 4
: If set, bits 5 and 6 specify accessed and dirty bits.

bit 5
: The page has been accessed (only valid if bit 4 is set).

bit 6
: The page has been written (is dirty). Only valid if bit 4 is set.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_page_size

Syntax

```
#include <dpmi.h>

int __dpmi_get_page_size(unsigned long *_size);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0604

Fills in the page size.

Return Value

-1 on error (16-bit host), else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_processor_exception_handler_vector

Syntax

```
#include <dpmi.h>
```

```
int __dpmi_get_processor_exception_handler_vector(
int _vector, __dpmi_paddr *_address
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0202

This function gets the current protected-mode exception handler (not interrupts) for the exception _vector. It will return a selector:offset pair in the members of the _address variable.

Return Value

-1 on error (invalid vector), else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_protected_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int __dpmi_get_protected_mode_interrupt_vector(int _vector,
__dpmi_paddr *_address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0204

This function gets the address of the current protected mode interrupt (not exception) handler. It returns a selector:offset pair.

Return Value

Zero. This always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_raw_mode_switch_addr

Syntax

```
#include <dpmi.h>

int __dpmi_get_raw_mode_switch_addr(__dpmi_raddr *_rm,
__dpmi_paddr *_pm);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0306

Read the spec for more info.

Return Value

Zero. This always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_real_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int __dpmi_get_real_mode_interrupt_vector(int _vector,
__dpmi_raddr *_address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0200

This function stores the real-mode interrupt vector address in _address. This is the same as the DOS get vector call, and returns a real-mode segment:offset pair.

Bits [31:8] in the vector number are silently ignored.

Return Value

Zero. This function always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_segment_base_address

Syntax

```
#include <dpmi.h>

int __dpmi_get_segment_base_address(int _selector,
unsigned long *_addr);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0006

The physical base address of the selector is stored in *addr.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned long addr;
if (__dpmi_get_segment_base_address(selector, &addr))
...
```

__dpmi_get_segment_limit

Syntax

```
#include <dpmi.h>
```

```
unsigned __dpmi_get_segment_limit(int _selector);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

Return Value

The limit of the segment, as returned by the `lsl` opcode.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_selector_increment_value

Syntax

```
#include <dpmi.h>

int __dpmi_get_selector_increment_value(void);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0003

Return Value

The value to add to each selector allocated by __dpmi_allocate_ldt_descriptors to get the next one.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_state_of_debug_watchpoint

Syntax

```
#include <dpmi.h>

int __dpmi_get_state_of_debug_watchpoint(unsigned long _handle,
int *_status);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0b02

Gets the state of the watchpoint. Pass handle, fills in status (0=not encountered, 1=encountered).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_state_save_restore_addr

Syntax

```
#include <dpmi.h>

int __dpmi_get_state_save_restore_addr(__dpmi_raddr *_rm,
```

```
__dpmi_paddr *_pm);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0305

Read the spec for info.

Return Value

The number of bytes required to save state.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_vendor_specific_api_entry_point

Syntax

```
#include <dpmi.h>

int __dpmi_get_vendor_specific_api_entry_point(char *_id,
__dpmi_paddr *_api);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0a00

Look up a vendor-specific function, given the *name* of the function.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_version

Syntax

```
#include <dpmi.h>

int __dpmi_get_version(__dpmi_version_ret *_ret);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0400

Fills in version information. The flags are as follows:

```
---- ---X = 0=16-bit host 1=32-bit host
---- --X- = 0=V86 used for reflected ints, 1=real mode
---- -X-- = 0=no virtual memory, 1=virtual memory supported
```

The cpu is 2=80286, 3=80386, 4=80486, etc.

DPMI 0.9 returns major=0 and minor=0x5a.

Return Value

Zero. This always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_get_virtual_interrupt_state

Syntax

```
#include <dpmi.h>

int __dpmi_get_virtual_interrupt_state(void);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0902

Return Value

This function returns the current interrupt flag (1=enabled).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_install_resident_service_provider_callback

Syntax

```
#include <dpmi.h>

int __dpmi_install_resident_service_provider_callback(
__dpmi_callback_info *info
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0c00 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function installs a resident service provider callback and declares an intent to provide resident protected-mode services after terminating with a call to `__dpmi_terminate_and_stay_resident` (See __dpmi_terminate_and_stay_resident).

The various members of info should be filled as follows:

`data16`
: An 8-byte descriptor for the 16-bit data segment.

`code16`
: An 8-byte descriptor for the 16-bit code segment (zeros if not supported).

`ip`
: A 16-bit offset of the 16-bit callback procedure.

`data32`
: An 8-byte descriptor for 32-bit data segment.

`code32`
: An 8-byte descriptor for 32-bit code segment (zeros if not supported).

`eip`

A 32-bit offset of the 32-bit callback procedure.

See __dpmi_get_descriptor, for the details about the layout of the 8-byte segment descriptor.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_int

Syntax

```
#include <dpmi.h>

int __dpmi_int(int _vector, __dpmi_regs *_regs);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0300

This function performs a software interrupt in real mode after filling in *most* the registers from the given structure. %ss, %esp, and %eflags are automatically taken care of, unlike See __dpmi_simulate_real_mode_interrupt.

The following variables can be used to tune this function. By default, these variables are all zero.

`__dpmi_int_ss`
`__dpmi_int_sp`
`__dpmi_int_flags`
These hold the values stored in the appropriate field in the `__dpmi_regs` structure.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_lock_linear_region

Syntax

```
#include <dpmi.h>

int __dpmi_lock_linear_region(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0600

This function locks virtual memory, to prevent page faults during hardware interrupts. Pass address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_map_conventional_memory_in_memory_block

Syntax

```
#include <dpmi.h>

int __dpmi_map_conventional_memory_in_memory_block(
__dpmi_meminfo *info, unsigned long physaddr
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0509 (DPMI 1.0 only). Supported by CWSDPMI, but not by Windows.

This function maps conventional memory (even when virtualized) to virtual memory. Pass the handle, offset, and number of pages.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_map_device_in_memory_block

Syntax

```
#include <dpmi.h>

int __dpmi_map_device_in_memory_block(__dpmi_meminfo *info,
unsigned long *physaddr);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0508 (DPMI 1.0 only). Supported by CWSDPMI, but not by Windows.

This function maps a physical address range to virtual memory. Pass the handle, offset relative to the start of the block, and number of pages to map.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_mark_page_as_demand_paging_candidate

Syntax

```
#include <dpmi.h>

int __dpmi_mark_page_as_demand_paging_candidate(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0702

Advises the server that certain pages are unlikely to be used soon. Set address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_mark_real_mode_region_as_pageable

Syntax

```
#include <dpmi.h>

int __dpmi_mark_real_mode_region_as_pageable(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0602

This function advises the host that the given pages are suitable for page-out. Pass address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_physical_address_mapping

Syntax

```
#include <dpmi.h>

int __dpmi_physical_address_mapping(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0800

Maps a physical device (like a graphics buffer) to linear memory. Fill in the physical address and size (in bytes). On return, the address is the linear address to use.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_relock_real_mode_region

Syntax

```
#include <dpmi.h>

int __dpmi_relock_real_mode_region(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0603

This function relocks the pages unlocked with See __dpmi_mark_real_mode_region_as_pageable. Pass address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_reset_debug_watchpoint

Syntax

```
#include <dpmi.h>

int __dpmi_reset_debug_watchpoint(unsigned long _handle);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0b03

Resets a watchpoint given its handle in _handle.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_resize_dos_memory

Syntax

```
#include <dpmi.h>

int __dpmi_resize_dos_memory(int _selector, int _newpara,
int *_ret_max);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0102

This function resizes a dos memory block. Remember to pass the selector, and not the segment. If this call fails, _ret_max contains the largest number of paragraphs available.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_resize_linear_memory

Syntax

```
#include <dpmi.h>

int __dpmi_resize_linear_memory(__dpmi_meminfo *info, int commit);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0505 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function resizes a memory block. Pass the handle and new size. Bit 0 of commit is 1 for committed pages; bit 1 is 1 to automatically update descriptors. It returns a new handle and base address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_resize_memory

Syntax

```
#include <dpmi.h>

int __dpmi_resize_memory(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0503

This function changes the size of a virtual memory block. You must pass the handle and size fields. It may change the base address also; beware of debugging breakpoints and locked memory. It will return a new handle.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_segment_to_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_segment_to_descriptor(int _segment);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0002

This function returns a selector that maps to what the real-mode segment provided would have referenced. Warning: this is a scarce resource.

Return Value

-1 on error, else the selector.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
short video = __dpmi_segment_to_descriptor(0xa000);
movedata(_my_ds(), buffer, video, 0, 320*200);
```

__dpmi_serialize_on_shared_memory

Syntax

```
#include <dpmi.h>

int __dpmi_serialize_on_shared_memory(unsigned long handle, int flags);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0d02 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function serializes access to a shared memory block whose handle is given in handle. The bit-mapped variable flags defines the following bits:

bit 0
: If set, return immediately if serialization is unavailable. If cleared, the program is suspended until the serialization becomes available.

bit 1
: If set, perform shared serialization. If cleared, perform exclusive serialization.

bits 2-15
: Reserved (should be zero).

An exclusive serialization blocks *any* serialization attempts for the same memory block from other virtual machines. A shared serialization blocks only *exclusive* serialization attempts from other virtual machines.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_coprocessor_emulation

Syntax

```
#include <dpmi.h>

int __dpmi_set_coprocessor_emulation(int flags);
```

Description

Please refer to See DPMI Specification, for details on DPMI function call operation. Also see See DPMI Overview, for general information.

DPMI function AX = 0x0e01 (DPMI v1.0 only, but supported by most DPMI v0.9 servers, including CWSDPMI, Windows, and QDPMI).

This function sets the co-processor emulation state as specified by flags. The only two used bits in flags are:

`bit 0`
: If set, enable the co-processor. If reset, disable the co-processor.

`bit 1`

If set, the emulation of the floating-point instructions will be done by the calling application.

DJGPP programs using one of the provided emulators should generally call this function with an argument of 2. (The DJGPP startup code does that automatically if no co-processor is detected.)

Return Value

-1 on errors, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_debug_watchpoint

Syntax

```
#include <dpmi.h>

int __dpmi_set_debug_watchpoint(__dpmi_meminfo *_info, int _type);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0b00

Set a debug breakpoint. Type is 0 for execute, 1 for write, and 2 for access. Fill in address and size (1,2,4 bytes). Server fills in handle.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_descriptor

Syntax

```
#include <dpmi.h>

int __dpmi_set_descriptor(int _selector, void *_buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000c

This function sets the parameters of the selector _selector by copying the contents of the 8-byte buffer pointed to by _buffer into the LDT entry of the selector's descriptor. See __dpmi_get_descriptor, for the description of the contents of the 8-byte buffer.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_descriptor_access_rights

Syntax

```
#include <dpmi.h>
```

```
int __dpmi_set_descriptor_access_rights(int _selector, int _rights);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0009

This sets the access rights of _selector to _rights.

The meaning of the individual bit fields of _rights is described below. For more details, please refer to See __dpmi_get_descriptor.

```
---- ---- ---- ---X = 0=not accessed, 1=accessed
---- ---- ---- --X- = data: 0=read, 1=r/w; code: 1=readable
---- ---- ---- -X-- = data: 0=expand-up, 1=expand-down;
code: 0=non-conforming
---- ---- ---- X--- = 0=data, 1=code
---- ---- ---1 ---- = must be 1
---- ---- -XX- ---- = priviledge level (must equal CPL)
---- ---- X--- ---- = 0=absent, 1=present
---X ---- ---- ---- = available for the user
--0- ---- ---- ---- = must be 0
-X-- ---- ---- ---- = 0=16-bit 1=32-bit
X--- ---- ---- ---- = 0=byte granular (small) 1=page-granular (big)
```

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_extended_exception_handler_vector_pm

Syntax

```
#include <dpmi.h>

int __dpmi_set_extended_exception_handler_vector_pm(
int vector, __dpmi_paddr *address
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0212 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function installs a handler for protected-mode exceptions.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_extended_exception_handler_vector_rm

Syntax

```
#include <dpmi.h>

int __dpmi_set_extended_exception_handler_vector_rm(
int vector, __dpmi_paddr *address
```

```
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0213 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function installs a handler for real-mode exceptions.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_multiple_descriptors

Syntax

```
#include <dpmi.h>

int __dpmi_set_multiple_descriptors(int count, void *buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x000f (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function sets multiple descriptors. Buffer usage is like in `__dpmi_get_multiple_descriptors` (See __dpmi_get_multiple_descriptors), but the caller fills in everything before calling.

Return Value

Returns count if successful, the negative of the number of descriptors set if failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_page_attributes

Syntax

```
#include <dpmi.h>

int __dpmi_set_page_attributes(__dpmi_meminfo *info, short *buffer);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0507 (DPMI 1.0 only). Supported by CWSDPMI, but not by Windows.

This function sets attributes of a number of pages. Pass handle in `info->handle`, offset within block in `info->address`, and number of pages in `info->count`. buffer points to an array of 16-bit words which specify the new attributes. See __dpmi_get_page_attributes, for the definition of the page attribute word.

The DJGPP startup code calls this function to uncommit the so-called null page, the first 4KB of the program's address space. This causes `NULL` pointer dereferences, a frequent programmatic error, to trigger a Page Fault exception, rather than go unnoticed.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_processor_exception_handler_vector

Syntax

```
#include <dpmi.h>

int __dpmi_set_processor_exception_handler_vector(
int _vector, __dpmi_paddr *_address
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0203

This function installs a handler for protected mode exceptions (not interrupts). You must pass a selector:offset pair.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_protected_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int __dpmi_set_protected_mode_interrupt_vector(int _vector,
__dpmi_paddr *_address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0205

This function installs a protected-mode interrupt (not exception) handler. You must pass a selector:offset pair. Hardware interrupts will always be reflected to protected mode if you install a handler. You must explicitely `sti` before `iret` because `iret` won't always restore interrupts in a virtual environment.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_real_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int __dpmi_set_real_mode_interrupt_vector(int _vector,
__dpmi_raddr *_address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0201

This function sets a real-mode interrupt vector. You must pass a segment:offset pair, not a selector.

Bits [31:8] in the vector number are silently ignored.

Return Value

Zero. This function always works.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_segment_base_address

Syntax

```
#include <dpmi.h>

int __dpmi_set_segment_base_address(int _selector, unsigned _address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0007

This function sets the base address of the _selector to _address.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_set_segment_limit

Syntax

```
#include <dpmi.h>

int __dpmi_set_segment_limit(int _selector, unsigned _address);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0008

This function sets the highest valid address in the segment referenced by _selector. For example, if you pass 0xfffff, the highest valid address is 0xfffff. Note: if you pass a number <= 64K, the segment changes to "non-big", and may cause unexpected problems. Limits for segments larger than 1MB must have their low 12 bits set.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_simulate_real_mode_interrupt

Syntax

```
#include <dpmi.h>

int __dpmi_simulate_real_mode_interrupt(int _vector,
__dpmi_regs *_regs);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0300

This function performs a software interrupt in real mode after filling in *all* the registers from the given structure. You *must* set %ss, %esp, and %eflags to valid real-mode values or zero, unlike See __dpmi_int.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_simulate_real_mode_procedure_iret

Syntax

```
#include <dpmi.h>

int __dpmi_simulate_real_mode_procedure_iret(__dpmi_regs *_regs);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0302

This function switches to real mode, filling in *all* the registers from the structure. ss:sp and flags must be valid or zero. The called function must return with an `iret`.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_simulate_real_mode_procedure_retf

Syntax

```
#include <dpmi.h>

int __dpmi_simulate_real_mode_procedure_retf(__dpmi_regs *_regs);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0301

This function switches to real mode with *all* the registers set from the structure, including cs:ip. The function called should return with a `retf`. ss:sp and flags must be set to valid values or zero.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_simulate_real_mode_procedure_retf_stack

Syntax

```
#include <dpmi.h>

int __dpmi_simulate_real_mode_procedure_retf_stack(
__dpmi_regs *_regs,
int stack_words_to_copy, const void *stack_data
);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0301

This function switches to real mode with *all* the registers set from the structure, including cs:ip. The function called should return with a `retf`. ss:sp and flags must be set to valid values or zero.

You may optionally specify data to be copied to the real-mode stack, to pass arguments to real-mode procedures with stack-based calling conventions. If you don't want to copy data to the real mode stack, pass 0 for stack_words_to_copy, and `NULL` for stack_bytes.

Note that the amount of stack data to be copied should be given in units of 16-bit words, not in bytes. This is defined by the underlying DPMI function.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_terminate_and_stay_resident

Syntax

```
#include <dpmi.h>

int __dpmi_terminate_and_stay_resident(int return_code,
int paragraphs_to_keep);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0c01 (DPMI 1.0 only). Not supported by CWSDPMI and Windows.

This function terminates the calling program, but leaves it resident in memory. return_code specifies which value to return to the OS. paragraphs_to_keep specifies the number of paragraphs of DOS (conventional) memory to keep; it should be either zero or 6 or more. Note that any protected-mode memory remains allocated to the program unless explicitly freed before calling this function.

The calling program **must** call the function `__dpmi_install_resident_service_provider_callback` before this one, otherwise it will be terminated instead of going TSR. See __dpmi_install_resident_service_provider_callback.

Return Value

This call does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_unlock_linear_region

Syntax

```
#include <dpmi.h>

int __dpmi_unlock_linear_region(__dpmi_meminfo *_info);
```

Description

Please refer to the DPMI Specification (See DPMI Specification) for details on DPMI function call operation. Also see the DPMI Overview (See DPMI Overview) for general information.

DPMI function AX = 0x0601

This function unlocks virtual memory. Pass address and size (in bytes).

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__dpmi_yield

Syntax

```
#include <dpmi.h>

void __dpmi_yield(void);
```

Description

`__dpmi_yield` calls function 1680h of the interrupt 2Fh, which tells the task manager in a multitasking environment that the calling program doesn't need the rest of its time slice. The task manager will then preempt the calling program and switch to another task that is ready to run.

This function should be called in busy-wait loops, like when a program waits for user input via keyboard, after it finds the keyboard buffer empty, to enhance overall performance in a multitasking environment.

Return Value

None. If the call isn't supported by the environment, like when running on plain DOS, `errno` is set to `ENOSYS`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

DTTOIF

Syntax

```
#include <dirent.h>

struct dirent *de;
mode_t file_mode = DTTOIF(de->d_type)
```

Description

This macro converts the `d_type` member of a `struct dirent` variable, as returned by `readdir` (See readdir) to an equivalent value of the `st_mode` member of a `struct stat` variable (See stat).

Note that the access rights are not set in the result returned by this macro. Only the file-type information is copied.

Return Value

The file's mode bits are returned. If the argument has the value `DT_UNKNOWN`, the result will be `S_IFREG`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No (see note 1)

Notes:

1. This macro is available on systems which support the `d_type` member in `struct dirent`.

dup

Syntax

```
#include <unistd.h>

int dup(int old_handle);
```

Description

This function duplicates the given file handle. Both handles refer to the same file and file pointer.

Return Value

The new file handle, or -1 if error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
do_file(dup(fileno(stdin)));
```

dup2

Syntax

```
#include <unistd.h>

int dup2(int existing_handle, int new_handle);
```

Description

This call causes new_handle to refer to the same file and file pointer as existing_handle. If new_handle is an open file, it is closed.

Return Value

The new handle, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
/* copy new file to stdin stream */
close(0);
dup2(new_stdin, 0);
close(new_stdin);
```

__dup_fd_properties

Syntax

```
#include <libc/fd_props.h>
```

```
void __dup_fd_properties(int existing_handle, int new_handle);
```

Description

Causes the new file descriptor new_handle to refer to the same `fd_properties` struct as existing_handle. This internal function is called by `dup` and `dup2`.

For more information, see `__set_fd_properties` (See __set_fd_properties) and `__clear_fd_properties` (See __clear_fd_properties).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_dxe_load

Syntax

```
#include <sys/dxe.h>

void *_dxe_load(char *dxe_filename);
```

Description

This function loads a dynamic executable image, whose file name is pointed to by dxe_filename, into memory and returns the entry point for the symbol associated with the image. The symbol may point to a structure or a function.

Return Value

0 on failure, the address of the loaded symbol on success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
static int (*add)(int a, int b);

add = _dxe_load("add.dxe");
if (add == 0)
printf("Cannot load add.dxe\n");
else
printf("Okay, 3 + 4 = %d\n", add(3,4));
```

DXE_macros

Syntax

```
#include <sys/dxe.h>

DXE_EXPORT_TABLE(name)
DXE_EXPORT_TABLE_AUTO(name)
DXE_EXPORT(symbol)
DXE_EXPORT_ASM(_symbol)
DXE_EXPORT_END
```

Description

These macros allows you to define a table of symbols that are going to be exported into subsequently loaded modules. If you use `DXE_EXPORT_TABLE_AUTO` instead of `DXE_EXPORT_TABLE` the table will be automatically registered with the dynamic loader during program startup (thus you don’t need to call `dlregsym(name)` manually).

```
DXE_DEMAND(name);
```

This macro declares the two functions that are present if you are statically linking against a dynamic library (see the dxe3gen section for details on static linking). Note that name should be in capitals with any illegal character converted to underscore. After declaring the module with the above macro, you can call the `dlload_NAME` and `dlunload_NAME` functions to dynamically load and unload the statically linked dynamic library.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ecvt

Syntax

```
#include <stdlib.h>

char * ecvt (double value, int ndigits, int *decpt, int *sign)
```

Description

This function converts the value into a null-terminated string, and returns a pointer to that string.

`ecvt` works exactly like `ecvtbuf` (See ecvtbuf), except that it generates the string in an internal static buffer which is overwritten on each call.

Return Value

A pointer to the generated string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ecvtbuf

Syntax

```
#include <stdlib.h>

char * ecvtbuf (double value, int ndigits, int *decpt, int *sign,
char *buf)
```

Description

This function converts its argument value into a null-terminated string of ndigits digits in buf. buf should have enough space to hold at least `ndigits + 1` characters.

The produced string in buf does *not* include the decimal point. Instead, the position of the decimal point relative to the beginning of buf is stored in an integer variable whose address is passed in decpt. Thus, if buf is returned as ‘‘1234’’ and *decpt as 1, this corresponds to a value of 1.234; if *decpt is -1, this corresponds to a value of 0.01234, etc.

The sign is also not included in buf’s value. If value is negative, `ecvtbuf` puts a nonzero value into the variable whose address is passed in sign; otherwise it stores zero in *sign.

The least-significant digit in buf is rounded.

`ecvtbuf` produces the string ‘‘NaN’’ if value is a NaN, and ‘‘Inf’’ or ‘‘Infinity’’ if value is an infinity (the longer form is produced when ndigits is 8 or more).

Return Value

A pointer to buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

char vbuf[20];
int esign, edecpt;

ecvtbuf (M_PI, 5, &edecpt, &esign, buf)
/* This will print " 31416". */
printf ("%c%s", esign ? '-' : ' ', buf);
```

edi_init

Syntax

```
#include <debug/dbgcom.h>

void edi_init (jmp_buf start_state);
```

Description

This function is part of the DJGPP debugging support. It should be called after a call to `v2loadimage` (See v2loadimage) which loads an executable program as a debuggee. `edi_init` then takes care of initializing the data structures which need to be set before the debugger can set breakpoints and run the debuggee.

The argument start_state is usually set by a preceding call to `v2loadimage`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (v2loadimage (exec_file, cmdline, start_state))
{
printf ("Load failed for image %s\n", exec_file);
exit (1);
}

edi_init (start_state);
```

enable

Syntax

```
#include <dos.h>

int enable(void);
```

Description

This function enables interrupts.

See disable.

Return Value

Returns nonzero if the interrupts were already enabled, zero if they had been disabled before this call.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int ints_were_enabled;

ints_were_enabled = enable();
. . . do some stuff . . .
if (!ints_were_enabled)
```

```
disable();
```

endgrent

Syntax

```
#include <grp.h>

void endgrent(void);
```

Description

This function should be called after all calls to `getgrent`, `getgrgid`, or `getgrnam`.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See getgrent.

endmntent

Syntax

```
#include <mntent.h>

int endmntent(FILE *filep);
```

Description

This function should be called after the last call to `getmntent` (See getmntent).

Return Value

This function always returns one.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

endpwent

Syntax

```
#include <pwd.h>

void endpwent(void);
```

Description

This function should be called after the last call to getpwent (See getpwent).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

errno

Syntax

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

```
extern int errno;
```

Description

This variable is used to hold the value of the error of the last function call. The value might be one of the following:

`0`
: No Error. Library functions never set `errno` to zero, but the startup code does that just before calling `main` (this is ANSI C requirement).

`1`
: EDOM -- Numerical input to a function is out of range.

`2`
: ERANGE -- Numerical output of a function is out of range.

`3`
: E2BIG -- Argument list too long. `system` and the functions from the `spawn` family assign this to `errno` when the command line is too long (longer than 126-character limit when invoking non-DJGPP programs, or longer than the transfer buffer size when invoking DJGPP programs).

`4`
: EACCES -- Permission denied. Attempt to write to a read-only file, or remove a non-empty directory, or open a directory as if it were a file, etc. In essence, it's a DOS way of saying "You can't do that, but I'm too stupid to know why."

`5`
: EAGAIN -- Resource temporarily unavailable, try again later. Almost never used in DJGPP, except when DOS returns error code 3Dh ("network print queue full") 81h (NetWare4 "CWait children still running") or 9Bh (NetWare4 "unable to create another TCB").

`6`
: EBADF -- Bad file descriptor: an invalid file handle passed to a library function.

`7`
: EBUSY -- Resource busy. Attempt to remove current directory (including current directory on another drive), or when a networked resource, such as a drive, is in use by another process.

`8`
: ECHILD -- No child processes. Returned by `wait` and `waitpid`, and by NetWare-related calls.

`9`
: EDEADLK -- Resource deadlock avoided. Never used in DJGPP.

`10`
: EEXIST -- File exists. Returned by `open` and `mkdir` when a file or directory by that name already exists.

`11`
: EFAULT -- Bad address. A function was passed a bad pointer (like a `NULL` pointer).

`12`
: EFBIG -- File too large. Never used in DJGPP.

`13`
: EINTR -- Interrupted system call. `system` and the functions of the `spawn` family use that when the child program was interrupted by **Ctrl-C**. Also, when DOS returns the "fail on INT 24h" error code.

`14`
: EINVAL -- Invalid argument. Any case when any argument to a library function is found to be invalid. Examples include invalid drive number, "." or ".." as one of the arguments to `rename`, syntax errors in the command line passed to `system`, etc.

`15`
: EIO -- Input or output error. Low-level error in I/O operation, like bad disk block, damaged FAT, etc.

`16`
: EISDIR -- Is a directory: an attempt to do something with a directory which is only allowed with regular

files. DOS usually returns `EACCES` in these cases, but DJGPP sometimes assigns `EISDIR` to `errno`, like when `rename` is called to move a regular file over a directory, or when `system` or one of the `spawn*` functions are passed a name of a directory instead of an executable program.

17
EMFILE -- Too many open files in system (no more handles available). This usually means that the number specified by the `FILES=` directive in `CONFIG.SYS` is too small.

18
EMLINK -- Too many links. Not used in DJGPP (as DOS doesn't support hard links).

19
ENAMETOOLONG -- File name too long (longer than `FILENAME_MAX`, defined in `stdio.h`).

20
ENFILE -- Too many open files. Never used in DJGPP.

21
ENODEV -- No such device. Attempt to access an invalid drive, or an invalid operation for the type of drive.

22
ENOENT -- No such file or directory.

23
ENOEXEC -- Unable to execute file. Returned by `_dxe_load` (when the argument names a file that isn't a valid DXE), and by NetWare-related calls which run programs remotely.

24
ENOLCK -- No locks available. Returned when the DOS file-locking functions cannot lock more files (due to overflow of the sharing buffer).

25
ENOMEM -- Not enough memory. Note that, unlike your expectations, `malloc` does NOT set `errno` to `ENOMEM`; however, several library functions that use `malloc` will do that when it returns a `NULL` pointer.

26
ENOSPC -- No space left on drive. DOS usually doesn't return this error, but `write` and `_write` do this for it, when they detect a full disk condition.

27
ENOSYS -- Function not implemented. Any system call that isn't supported by the underlying OS, like an LFN function when running on plain DOS.

28
ENOTDIR -- Not a directory. DOS never returns this code, but some library functions, like `rename` and `_truename`, do that if they expect a valid directory pathname, but get either an invalid (e.g. empty) pathname or a file that is not a directory.

29
ENOTEMPTY -- Directory not empty. DOS never returns this code, but `rename` does, when it is called to move a directory over an existing non-empty directory.

30
ENOTTY -- Inappropriate I/O control operation. The termios functions set `errno` to this when called on a device that is not a TTY.

31
ENXIO -- No such device or address. An operation attempted to reference a device (not a disk drive) that is invalid, or non-existent, or access a disk drive that exists but is empty.

32
EPERM -- Operation not permitted. Examples include: sharing or file lock violations; denial of access to networked resources; expired password or illegal login attempts via a network; too many or duplicate network redirections; etc.

33
EPIPE -- Broken pipe: attempt to write to a pipe with nobody to read it. This never happens in DJGPP.

34
EROFS -- Read-only file system: attempt to write to a read-only disk. Unfortunately, DOS almost never returns this code.

35
ESPIPE -- Invalid seek: attempt to seek on a pipe. Never happens in DJGPP, except for NetWare-related operations, since pipes are simulated with regular files in MS-DOS, and therefore are always seekable.

36
ESRCH -- No such process. Not used in DJGPP.

37
EXDEV -- Improper link. An attempt to rename a file across drives or create a cross-device hardlink.

38
ENMFILE -- No more files. `findfirst` and `findnext` assign this to `errno` when they exhaust the files in the directory. `readdir` does that as well.

39
ELOOP -- Too many levels of symbolic links. Can be set virtually by any file handling function in library. Usually means encountered link loop (link1 -> link2, link2 -> link1).

40
EOVERFLOW -- Value too large. `filelength` can assign this to `errno` when a file's length is larger than 2^31-2 (See filelength). `lfilelength` can assign this to `errno` when a file's length is larger than 2^63-1 (See lfilelength).

41
EILSEQ -- Invalid or incomplete multibyte or wide character.

See perror.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

exec*

Syntax

```
#include <unistd.h>

int execl(const char *path, const char *argv0, ...);
int execle(const char *path, const char *argv0, ...
/*, char *const envp[] */);
int execlp(const char *path, const char *argv0, ...);
int execlpe(const char *path, const char *argv0, ...
/*, char *const envp[] */);

int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *path, char *const argv[]);
int execvpe(const char *path, char *const argv[], char *const envp[]);
```

Description

These functions operate by calling `spawn*` with a type of `P_OVERLAY`. Refer to `spawn*` (See spawn*) for a full description.

Return Value

If successful, these functions do not return. If there is an error, these functions return -1 and set `errno` to indicate the error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
execlp("gcc", "gcc", "-v", "hello.c", 0);
```

__exit

Syntax

```
#include <unistd.h>

void __exit(int exit_code);
```

Description

This is an internal library function which exits the program, returning exit_code to the calling process. No additional processing is done, and any `atexit` functions are not called. Since hardware interrupts are not unhooked, this can cause crashes after the program exits. This function is normally called only by `_exit`; do *not* call it directly.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

_exit

Syntax

```
#include <unistd.h>

void _exit(int exit_code);
```

Description

This function exits the program, returning exit_code to the calling process. No additional processing (such as closing file descriptors or calls to the static destructor functions) is done, and any `atexit` functions are not called; only the hardware interrupt handlers are unhooked, to prevent system crashes e.g. after a call to `abort`. This function is normally called only by `exit` and `abort`.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_Exit

Syntax

```
#include <stdlib.h>

void _Exit(int exit_code);
```

Description

This function exits the program, returning exit_code to the calling process. No additional processing (such as closing file descriptors or calls to the static destructor functions) is done, and any `atexit` functions are not called; only the hardware interrupt handlers are unhooked, to prevent system crashes.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99 (see note 1); not C89 1003.1-2001; not 1003.2-1992

Notes:

1. Depending on the implementation, `_Exit` may do the additional processing described above.

Example

```
if (argc < 4)
{

print_usage();
_Exit(1);
}
```

exit

Syntax

```
#include <stdlib.h>

void exit(int exit_code);
```

Description

This function exits the program, returning exit_code to the calling process. Before exiting, all open files are closed and all `atexit` and `on_exit` requests are processed.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (argc < 4)
{

print_usage();
exit(1);
}
```

exp

Syntax

```
#include <math.h>

double exp(double x);
```

Description

This function computes the exponential of x, e^x, where **e** is the base of the natural system of logarithms, approximately 2.718281828.

Return Value

e to the x power. If the value of x is finite, but so large in magnitude that its exponential cannot be accurately represented by a `double`, the return value is the nearest representable `double` (possibly, an `Inf`), and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or zero, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

exp10

Syntax

```
#include <math.h>

double exp10(double x);
```

Description

This function computes 10 to the power of x, 10^x.

Return Value

10 to the x power. If the value of x is finite, but so large in magnitude that 10^x cannot be accurately represented by a `double`, the return value is the nearest representable `double` (possibly, an `Inf`), and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or zero, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

exp2

Syntax

```
#include <math.h>

double exp2(double x);
```

Description

This function computes 2 to the power of x, 2^x.

Return Value

2 to the x power. If the value of x is finite, but so large in magnitude that 2^x cannot be accurately represented by a `double`, the return value is is the nearest representable `double` (possibly, an `Inf`), and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or zero, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

expm1

Syntax

```
#include <math.h>

double expm1(double x);
```

Description

This function computes the value of e^x - 1, the exponential of x minus 1, where **e** is the base of the natural system of logarithms, approximately 2.718281828. The result is more accurate than `exp(x) - 1` for small values of x, where the latter method would lose many significant digits.

Return Value

e raised to the power x, minus 1. If the value of x is finite, but so large that its exponent would overflow a `double`, the return value is `Inf`, and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or -1, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fabs

Syntax

```
#include <math.h>

double fabs(double x);
```

Description

This function computes the absolute value of its argument x.

Return Value

x if x is positive, else -x. Note that in this context, +0.0 is positive and -0.0 is negative. Infinities and `NaN`s are returned unchanged, except for the sign.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

_far*

Syntax

```
#include <sys/farptr.h>

unsigned char _farpeekb(unsigned short selector, unsigned long offset);
unsigned short _farpeekw(unsigned short selector, unsigned long offset);
unsigned long _farpeekl(unsigned short selector, unsigned long offset);

void _farpokeb(unsigned short sel, unsigned long off,
unsigned char val);
void _farpokew(unsigned short sel, unsigned long off,
unsigned short val);
void _farpokel(unsigned short sel, unsigned long off,
unsigned long val);

void _farsetsel(unsigned short selector);
unsigned short _fargetsel(void);

void _farnspokeb(unsigned long offset, unsigned char value);
void _farnspokew(unsigned long offset, unsigned short value);
void _farnspokel(unsigned long offset, unsigned long value);

unsigned char _farnspeekb(unsigned long offset);
unsigned short _farnspeekw(unsigned long offset);
unsigned long _farnspeekl(unsigned long offset);
```

Description

These functions provide the equivalent functionality of "far pointers" to peek or poke an absolute memory addresses, even though gcc doesn't understand the keyword "far". They come in handy when you need to access memory-mapped devices (like VGA) or some address in lower memory returned by a real-mode service. These functions are provided as inline assembler functions, so when you optimize your program they reduce to only a few opcodes (only one more than a regular memory access), resulting in very optimal code.

The first two groups of functions take a selector and an offset. This selector is *not* a dos segment. If you want to access dos memory, pass _go32_info_block.selector_for_linear_memory (or just _dos_ds, which is defined in the include file go32.h) as the selector, and seg*16+ofs as the offset. For functions which poke the memory, you should also provide the value to put there.

The last two groups assume that you've used `_farsetsel` to specify the selector. You should avoid making any function calls between `_farsetsel` and using these other functions, unless you're absolutely sure that they won't modify that selector. This allows you to optimize loops by setting the selector once outside the loop, and using the shorter functions within the loop. You can use `_fargetsel` if you want to temporary change the selector with `_farsetsel` and restore it afterwards.

Return Value

Functions which peek the address return the value at given address. `_fargetsel` returns the current selector.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fchdir

Syntax

```
#include <unistd.h>

int fchdir(int fd);
```

Description

This function changes the current directory to the directory described by the file descriptor fd.

Return Value

Zero on success, else nonzero and errno set if error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int fd;

fd = open("dir", O_RDONLY);
fchdir(fd);
```

fchmod

Syntax

```
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

Description

This function changes the mode (writable or write-only) of the file opened under the file descriptor fd. The value of mode can be a combination of one or more of the `S_I*` constants described in the description of the `chmod` function (See chmod).

Some `S_I*` constants are ignored for regular files:

- `S_I*GRP` and `S_I*OTH` are ignored, because DOS/Windows has no concept of ownership, so all files are considered to belong to the user;
- `S_IR*` are ignored, because files are always readable on DOS/Windows.

`fchmod` will always succeed for character devices, but the mode will be ignored.

`fchmod` may not be able to change the mode of files that have been opened using low-level methods. High-level methods for opening files include the `fopen` (See fopen) and `open` (See open) functions. Low-level methods include the `_open` (See _open) and `_dos_open` (See _dos_open) functions. In particular, redirected handles cannot have their mode changed with `fchmod`.

`fchmod` may also not work as expected under DOS. For instance, if a file is opened as read-write, then changed to read-only with `fchmod`, but then written to, then the mode will not be changed.

This function can be hooked by File System Extensions (See File System Extensions).

Return Value

Zero if the file exists and the mode was changed, else -1.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int fd;

fd = open("/tmp/dj.dat", O_RDWR);
fchmod(fd, S_IWUSR|S_IRUSR);
```

fchown

Syntax

```
#include <unistd.h>

int fchown(int fd, uid_t owner, gid_t group);
```

Description

This function changes the ownership of the open file specified by fd to the user ID owner and group ID group.

This function does almost nothing under MS-DOS: it just checks if the handle `fd` is valid. This function can be hooked by File System Extensions (See File System Extensions).

Return Value

This function returns zero if the handle is valid, non-zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fclose

Syntax

```
#include <stdio.h>

int fclose(FILE *file);
```

Description

This function closes the given file.

Return Value

Zero on success, else `EOF`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
FILE *f = fopen("data", "r");
fprintf(f, "Hello\n");
fclose(f);
```

fcntl

Syntax

```
#include <fcntl.h>

int fcntl (int fd, int cmd, ...);
```

Description

This function performs the operation specified by cmd on the file open on handle fd. The following operations are defined by the header `fcntl.h`:

`F_DUPFD`
: Returns a file handle that duplicates fd like `dup` does (See dup), except that `fcntl` also makes sure the returned handle is the lowest available handle greater than or equal to the integer value of the third argument.

`F_GETFD`
: Get the `FD_CLOEXEC` close-on-exec (a.k.a. no-inherit) status of fd. If the returned value has its least-significant bit set, the file will not be inherited by programs invoked by this process; otherwise, the file will remain open in the child processes.

  Note that only the first 20 handles can be passed to child processes by DOS/Windows; handles beyond that cannot be inherited. In addition, the stub loader of the child DJGPP program will forcibly close handles 19 and 18 (since otherwise it will be unable to read the COFF executable information and enter protected mode). Therefore, the current implementation always returns `FD_CLOEXEC` for handles 18 and above.

  For handles less than 18, the call will try to determine the status of the `O_NOINHERIT` flag for that file and will return either `FD_CLOEXEC` if the flag is set, or 0 if the flag is not set. If the status of the `O_NOINHERIT` flag cannot be determined, the call will return -1, setting `errno` to `ENOSYS`.

  The no-inherit bit can be set when the file is opened by using the `O_NOINHERIT` in the open flags; see See open.

`F_SETFD`
: Set or unset the close-on-exec flag for the handle fd using the LSB of the integer value supplied as the third argument. Since only the first 20 handles are passed to child programs, and since the stub loader of the child DJGPP program will forcibly close handles 19 and 18 (since otherwise it will be unable to read the COFF executable information and enter protected mode), the flag can only be set or unset on the first 18 handles. Attempts to set the flag for handles 18 or above will always return 0, and attempts to unset the flag for handles 18 or above will always return -1, setting `errno` to `ENOSYS`.

  For handles less than 18, the call will try to set or unset the `O_NOINHERIT` flag for that file and will return 0 if the flag is changed. If the `O_NOINHERIT` flag cannot be changed, the call will return -1, setting `errno` to `ENOSYS`.

`F_GETFL`
: Get the open mode and status flags associated with the handle fd. The flags are those supported by `open` and `creat` functions, like `O_RDONLY`, `O_APPEND`, etc.

  On Windows NT this cannot report the open mode correctly --- `O_RDONLY` is always returned.

`F_SETFL`
: Set the open mode and status flags associated with the handle fd. This fails in all but one case, and sets `errno` to `ENOSYS`, since DOS and Windows don't allow changing the descriptor flags after the file is open.

  The one allowed case is for `O_NONBLOCK`, since DJGPP doesn't support it anyway. That is, calls using `F_SETFL` will fail for all flag values **except** `O_NONBLOCK`.

```
#include <fcntl.h>

ret = fcntl(fd, F_SETFL, O_BINARY); /* This will fail, returning -1 */
/* and setting errno to ENOSYS. */

ret = fcntl(fd, F_SETFL, O_NONBLOCK); /* This will succeed */
/* returning 0. */
```

`F_GETLK`
: Return the lock structure that prevents obtaining the lock pointed to by the third argument, or set the `l_type` field of the lock structure to `F_UNLCK` if there is no obstruction. Currently, only the setting of the `l_type` field is provided. This call will not return values in the `struct flock` parameter identifying what lock parameters prevent getting the requested lock, since there is no way to obtain this information from DOS/Windows. If the lock cannot be obtained, -1 is returned and `errno` is set to the reason (which will be one of `EINVAL`, `EBADF`, `EACCES` or `ENOLCK`).

  Locking of directories is not supported.

`F_SETLK`
: Set or clear a file segment lock according to the structure pointed to by the third argument. The lock is set when `l_type` is `F_RDLCK` (shared lock request) or `F_WRLCK` (exclusive lock request), and the lock is

cleared when l_type is F_UNLCK. If the lock is already held, then this call returns -1 and sets errno to EACCES.

The F_RDLCK value for requesting a read lock is always treated as if it were F_WRLCK for a write lock.

This is because DOS/Win9x only supports one kind of lock, and it is the exclusive kind.

Locking of directories is not supported.

F_SETLKW
: Same as F_SETLK, but if the lock is blocked, the call will wait (using __dpmi_yield, see See __dpmi_yield) until it is unblocked and the lock can be applied. This call will **never exit** if the program making the call is the program which already owns the lock.

Locking of directories is not supported.

F_GETLK64
F_SETLK64
F_SETLKW64
: Each of these does exactly the same function as the non-"64" version, but the third argument must be of type struct flock64, which allows the l_start and l_len members to be long long int values. The current code will only use these long long int values modulo 2^32, which allows file locking positions up to 4 gigabytes minus 1. True 64-bit file locking is **not** supported.

The struct flock64 members l_start and l_len are declared to be of type offset_t, which is in turn typedef'ed to be a long long.

Locking of directories is not supported.

This function can be hooked by the Filesystem extensions, see See File System Extensions. If you don't want this, and you are calling fcntl with the F_DUPFD command, you should use dup2 instead, see See dup2.

Return Value

If an invalid or unsupported value is passed in cmd, or fd is an invalid file handle, the function returns -1 and sets errno to the appropriate value. Unsupported values of cmd cause ENOSYS to be stored in errno. If cmd is F_DUPFD, the function returns the new descriptor or -1 in case of a failure.

Lock requests which specify the open file's current EOF position as the value of l_start and zero as the l_len value will fail, returning -1 with errno set to EACCES.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. Contrary to Posix requirement, the handle returned by F_DUPFD shares the FD_CLOEXEC flag with fd (unless they are on different sides of the 20-handle mark), since DOS/Windows only maintain a single set of bits for all the handles associated with the same call to open.

Example

```
/* Save the handle in a way that it won't be passed
to child processes. */
int saved_fd = fcntl(fd, F_DUPFD, 20);

/* Set an advisory lock for the whole file. */
struct flock flock;
int retval, fd;

flock.l_type = F_RDLCK;
flock.l_whence = SEEK_SET;
flock.l_start = flock.l_len = 0;
errno = 0;
retval = fcntl(fd, F_SETLK, &flock);

/* Get the status of the lock we just obtained
```

```
(should return -1 with errno == EACCES). */
errno = 0;
retval = fcntl(fd, F_GETLK, &flock);

/* Release the lock. */
errno = 0;
flock.l_type = F_UNLCK;
retval = fcntl(fd, F_SETLK, &flock);

/* Get the status of the lock we just released
(should return 0). */
errno = 0;
flock.l_type = F_RDLCK;
retval = fcntl(fd, F_GETLK, &flock);

/* Try to set the O_BINARY flag on the open file
(should return -1 with errno == ENOSYS). */
errno = 0;
retval = fcntl(fd, F_SETFL, O_BINARY);

/* Set the O_NONBLOCK flag on the open file
(should return 0). */
errno = 0;
retval = fcntl(fd, F_SETFL, O_NONBLOCK);

/* Get the flags on the open file
(always returns 0). */
errno = 0;
retval = fcntl(fd, F_GETFL);
```

fcvt

Syntax

```
#include <stdlib.h>

char * fcvt (double value, int ndigits, int *decpt, int *sign)
```

Description

This function converts the value into a null-terminated string, and returns a pointer to that string.

`fcvt` works exactly like `fcvtbuf` (See fcvtbuf), except that it generates the string in an internal static buffer which is overwritten on each call.

Return Value

A pointer to the generated string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fcvtbuf

Syntax

```
#include <stdlib.h>

char * fcvtbuf (double value, int ndigits, int *decpt, int *sign,
char *buf)
```

Description

This function converts its argument value into a null-terminated string in buf with ndigits digits *to the right* of the decimal point. ndigits can be negative to indicate rounding to the left of the decimal point. buf should have enough space to hold at least `310+max(0,ndigits)` characters.

Note that, unlike `ecvtbuf` (See ecvtbuf), `fcvtbuf` only counts the digits *to the right* of the decimal point. Thus, if value is 123.45678 and ndigits is 4, then `ecvtbuf` will produce ‘‘1235’’, but `fcvtbuf` will produce ‘‘1234568’’ (and *decpt will be 3 in both cases).

The produced string in buf does *not* include the decimal point. Instead, the position of the decimal point relative to the beginning of buf is stored in an integer variable whose address is passed in decpt. Thus, if buf is returned as ‘‘1234’’ and *decpt as 1, this corresponds to a value of 1.234; if *decpt is -1, this corresponds to a value of 0.01234, etc.

The sign is also not included in buf’s value. If value is negative, `ecvtbuf` puts a nonzero value into the variable whose address is passed in sign; otherwise it stores zero in *sign.

The least-significant digit in buf is rounded.

`ecvtbuf` produces the string ‘‘NaN’’ if value is a NaN, and ‘‘Inf’’ or ‘‘Infinity’’ if value is an infinity (the longer form is produced when ndigits is 8 or more).

Return Value

A pointer to buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

char vbuf[20];
int fsign, fdecpt;

fcvtbuf (M_PI, 5, &fdecpt, &fsign, buf)
/* This will print " 314159". */
printf ("%c%s", fsign ? ’-’ : ’ ’, buf);
```

fdopen

Syntax

```
#include <stdio.h>

FILE *fdopen(int fd, const char *mode);
```

Description

This function opens a stream-type file that uses the given fd file, which must already be open. The file is opened with the modes specified by mode, which is the same as for `fopen`. See fopen.

Return Value

The newly created `FILE *`, or `NULL` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
FILE *stdprn = fdopen(4, "w");
```

feof

Syntax

```
#include <stdio.h>

int feof(FILE *file);
```

Description

This function can be used to indicate if the given file is at the end-of-file or not.

Return Value

Nonzero at end-of-file, zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
while (!feof(stdin))
gets(line);
```

ferror

Syntax

```
#include <stdio.h>

int ferror(FILE *file);
```

Description

This function can be used to indicate if the given file has encountered an error or not. See clearerr.

Return Value

Nonzero for an error, zero otherwize.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (ferror(stdin))
exit(1);
```

fflush

Syntax

```
#include <stdio.h>

int fflush(FILE *file);
```

Description

If file is not a `NULL` pointer, this function causes any unwritten buffered data to be written out to the given file. This is useful in cases where the output is line buffered and you want to write a partial line.

If file is a `NULL` pointer, `fflush` writes any buffered output to all files opened for output.

Note that `fflush` has no effect for streams opened for reading only. Also note that the operating system can further buffer/cache writes to disk files; a call to `fsync` (See fsync) or `sync` (See sync) is typically required to actually deliver data to the file(s).

Return Value

Zero on success, -1 on error. When called with a `NULL` pointer, -1 will be returned if an error happened while flushing some of the streams (but `fflush` will still try to flush all the rest before it returns).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
printf("Enter value : ");
fflush(stdout);
scanf(result);
```

ffs

Syntax

```
#include <string.h>

int ffs(int _mask);
```

Description

This function find the first (least significant) bit set in the input value.

Return Value

Bit position (1..32) of the least significant set bit, or zero if the input value is zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ffs(0) = 0
ffs(1) = 1
ffs(5) = 1
ffs(96) = 6
```

fgetc

Syntax

```
#include <stdio.h>

int fgetc(FILE *file);
```

Description

Returns the next character in the given file as an unsigned char.

Return Value

The given char (value 0..255) or `EOF` at end-of-file.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int c;
while((c=fgetc(stdin)) != EOF)
fputc(c, stdout);
```

fgetgrent

Syntax

```
#include <grp.h>

struct group *fgetgrent(FILE *file);
```

Description

This function, in MS-DOS, is exactly the same as `getgrent` (See getgrent).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fgetpos

Syntax

```
#include <stdio.h>

int fgetpos(FILE *file, fpos_t *offset);
```

Description

This function records the current file pointer for file, for later use by `fsetpos`.

See fsetpos. See ftell.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

fgets

Syntax

```
#include <stdio.h>

char *fgets(char *buffer, int maxlength, FILE *file);
```

Description

This function reads as much of a line from a file as possible, stopping when the buffer is full (maxlength-1 characters), an end-of-line is detected, or `EOF` or an error is detected. It then stores a `NULL` to terminate the string.

Return Value

The address of the buffer is returned on success, if `EOF` is encountered before any characters are stored, or if an error is detected, `NULL` is returned instead.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char buf[100];
while (fgets(buf, 100, stdin))
fputs(buf, stdout);
```

File System Extensions

Description

The File System Extensions are a part of the lowest level of I/O operations in the C runtime library of DJGPP. These extensions are provided to allow for cases where Unix uses a file descriptor to access such items as serial ports, memory, and the network, but DOS does not. It allows a set of functions (called an *extension*) to gain control when one of these low-level functions is called on a file descriptor set up by the extension.

Each extension must provide one or two handler functions. All handler functions take the same arguments:

```
int function(__FSEXT_Fnumber func_number, int *rv, va_list args);
```

The func_number identifies which function is to be emulated. The file `<sys/fsext.h>` defines the function numbers as follows:

`__FSEXT_nop`
: A no-op. This is currently unused by the library functions.

`__FSEXT_open`
: An open handler (See _open). This is called just before the library is about to issue the DOS OpenFile call on behalf of your program.

  If `_open` was called from the library function `open`, then the file name passed to the handler will have either all its symlink components resolved or will refer to a symlink file (i.e.: all directory symlinks will be resolved), depending on whether the `O_NOLINK` was passed to `open` (See open).

  Do not use this extension to emulate symlinks. Use `__FSEXT_readlink` handler instead.

`__FSEXT_creat`
: A create handler (See _creat, and See _creatnew). Called when a file needs to be created. Note that the handler should both create the ‘‘file’’ and open it.

  If `_creat` or `_creatnew` were called from the library functions `open` or `creat`, then the file name passed to the handler will have all its symlink components resolved.

`__FSEXT_read`
: A read handler (See _read). Called when data should be read from a ‘‘file’’.

`__FSEXT_write`
: A write handler (See write, and See _write). Called to write data to a ‘‘file’’. On ‘‘text’’ files it receives the *original* (unconverted) buffer.

`__FSEXT_ready`
: A ready handler. It is called by `select` library function (See select) when it needs to know whether a handle used to reference the ‘‘file’’ is ready for reading or writing, or has an error condition set. The handler should return an OR’ed bit mask of the following bits (defined on `<sys/fsext.h>`):

  `__FSEXT_ready_read`
  : The ‘‘file’’ is ready for reading.

  `__FSEXT_ready_write`
  : The ‘‘file’’ is ready for writing.

  `__FSEXT_ready_error`
  : The ‘‘file’’ has an error condition set.

`__FSEXT_close`
: A close handler (See _close). Called when the ‘‘file’’ should be closed.

`__FSEXT_fcntl`
: A file fcntl handler (See fcntl).

`__FSEXT_ioctl`
: A file ioctl handler (See ioctl (General description)).

`__FSEXT_lseek`
: A file lseek handler (See lseek). Here for backwards compatibility. Use `__FSEXT_llseek` instead. If you have a `__FSEXT_llseek` handler you don’t need a `__FSEXT_lseek` handler as `lseek` calls `llseek` internally.

`__FSEXT_llseek`
: A file llseek handler (See llseek).

`__FSEXT_link`
: A file link handler (See link). This is most relevant to file system extensions that emulate a directory structure.

  The source and destination file names are passed to the handler unchanged.

`__FSEXT_unlink`

A file unlink handler (See remove, and See unlink). This is most relevant to file system extensions that emulate a directory structure.

The file name passed to the handler will have all its directory symlinks resolved, so it may refer to a symlink file.

`__FSEXT_dup`
: A file dup handler (See dup). This is called when a new descriptor is needed to refer to an existing descriptor.

`__FSEXT_dup2`
: A file dup2 handler (See dup2). This is called when two different file descriptors are used to refer to the same open file.

`__FSEXT_stat`
: A file lstat handler (See lstat). This extension should provide information about stated file. If you provide this hook, function `stat` will be hooked too, as `stat` always calls `lstat`.

If the handler is called as a result of a `stat` call, then the file name passed to the handler will have all its symlinks resolved, so it will refer to a ''regular'' file. If the handler is called as result of a `lstat` call and not a `stat` call, then the file name passed to the handler will have all its directory symlinks resolved, so it may refer to a symlink file.

`__FSEXT_fstat`
: A file fstat handler (See fstat). The extension should fill in various status information about the emulated file.

`__FSEXT_readlink`
: A file readlink handler (See readlink). This extension should be used to provide support for symlinks in some other than DJGPP format.

The file name passed to the handler will have all its directory symlinks resolved, so it may refer to a symlink file.

`__FSEXT_symlink`
: A file symlink handler (See symlink). This extension should create symlinks in other than DJGPP symlink file format.

The source and destination file names are passed to the handler unchanged.

`__FSEXT_chmod`
: A file chmod handler (See chmod). This is called when the permissions are to be changed for a file.

The file name passed to the handler will have all its symlinks resolved.

`__FSEXT_chown`
: A file chown handler (See chown). This is called when the ownership is to be changed for a file.

The file name passed to the handler will have all its symlinks resolved.

`__FSEXT_fchmod`
: A file fchmod handler (See fchmod). This is called when the permissions are to be changed for an open file.

`__FSEXT_fchown`
: A file fchown handler (See fchown). This is called when the ownership is to be changed for an open file.

rv points to a temporary return value pointer. If the function is emulated by the handler, the return value should be stored here, and the handler should return a nonzero value. If the handler returns zero, it is assumed to have not emulated the call, and the regular DOS I/O function will happen. The args represent the arguments passed to the original function; these point to the actual arguments on the stack, so the emulation may choose to modify them and return zero to the regular function, which will then act on the modified arguments.

A normal extension would provide these parts:

- Some function to create a connection to the extension. This may be a custom function (such as `socket` for networking) or an extension to open (such as `/dev/ttyS0` to access the serial port).

- Initialization code that adds the open handler, if any.

- Overrides for the basic I/O functions, such as `read` and `write`. This is a single function in the extension that uses the function number parameter to select an extension function.

- The core functionality of the extension, if any.

Please note that the special Unix filenames `/dev/null` and `/dev/tty` are already mapped to the appropriate DOS names `NUL` and `CON`, respectively, so you don't need to write extensions for these.

Please note that the special Unix filenames `/dev/zero` and `/dev/full` can be made available by calling the functions `__install_dev_zero` (See __install_dev_zero) and `__install_dev_full` (See __install_dev_full) respectively, so you don't need to write extensions for these. These are implemented using File System Extensions.

Programs using the DJGPP debug support functions in `libdbg.a` may have problems using File System Extensions, because the debug support functions use a File System Extension to track the opening and closing of files. Only the `__FSEXT_open` and `__FSEXT_creat` calls will be passed to other File System Extensions by `libdbg.a`. In other words, only fairly trivial File System Extensions can be used in programs at the same time as the debug support functions.

__file_exists

Syntax

```
#include <unistd.h>

int __file_exists(const char *_fn);
```

Description

This function provides a fast way to ask if a given file exists. Unlike access(), this function does not cause other objects to get linked in with your program, so is used primarily by the startup code to keep minimum code size small.

Return Value

Zero if the file does not exist, nonzero if it does. Note that this is the opposite of what access() returns.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (__file_exists(fname))
process_file(fname);
```

__file_tree_walk

Syntax

```
#include <dir.h>

int __file_tree_walk(const char *dir,
int (*func)(const char *path,
const struct ffblk *ff));
```

Description

This function recursively descends the directory hierarchy which starts with dir. For each file in the hierarchy, `__file_tree_walk` calls the user-defined function func which is passed a pointer to a `NULL`-terminated character array in path holding the full pathname of the file, a pointer to a `ffblk` structure (See findfirst) ff with a DOS filesystem information about that file.

This function always visits a directory before any of its siblings. The argument dir must be a directory, or `__file_tree_walk` will fail and set `errno` to `ENOTDIR`. The directory dir itself is never passed to func.

The tree traversal continues until one of the following events:

(1) The tree is exhausted (i.e., all descendants of dir are processed). In this case, `__file_tree_walk` returns 0,

meaning a success.

(2) An invocation of func returns a non-zero value. In this case, `__file_tree_walk` stops the tree traversal and returns whatever func returned.

(3) An error is detected within `__file_tree_walk`. In that case, `ftw` returns -1 and sets `errno` (See errno) to a suitable value.

Return Value

Zero in case the entire tree was successfully traversed, -1 if `__file_tree_walk` detected some error during its operation, or any other non-zero value which was returned by the user-defined function func.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <stdlib.h>

int
ff_walker(const char *path, const struct ffblk *ff)
{

printf("%s:\t%lu\t", path, ff->ff_fsize);
if (ff->ff_attrib & 1)
printf("R");
if (ff->ff_attrib & 2)
printf("H");
if (ff->ff_attrib & 4)
printf("S");
if (ff->ff_attrib & 8)
printf("V");
if (ff->ff_attrib & 0x10)
printf("D");
if (ff->ff_attrib & 0x20)
printf("A");
printf("\n");

if (strcmp(ff->ff_name, "XXXXX") == 0)
return 42;
return 0;
}


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

if (argc > 1)
{
char msg[80];

sprintf(msg, "__file_tree_walk: %d",
__file_tree_walk(argv[1], ff_walker));
if (errno)
perror(msg);
else
puts(msg);
}
else
printf("Usage: %s dir\n", argv[0]);

return 0;
}
```

filelength

Syntax

```
#include <io.h>

long filelength(int fhandle);
```

Description

This function returns the size, in bytes, of a file whose handle is specified in the argument fhandle. To get the handle of a file opened by `fopen` (See fopen) or `freopen` (See freopen), you can use `fileno` macro (See fileno).

Return Value

The size of the file in bytes, or (if any error occured) -1L and `errno` set to a value describing the cause of the failure. If the file's length is larger than a 32-bit `unsigned int` can hold, `errno` will be set to `EOVERFLOW`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("Size of file to which STDIN is redirected is %ld\n",
filelength(0));
```

fileno

Syntax

```
#include <stdio.h>

int fileno(FILE *file);
```

Description

This function returns the raw file descriptor number that file uses for I/O.

Return Value

The file descriptor number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

findfirst

Syntax

```
#include <dir.h>

int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
```

Description

This function and the related `findnext` (See findnext) are used to scan directories for the list of files therein. The pathname is a wildcard that specifies the directory and files to search for (such as `subdir/*.c`), ffblk is a structure to hold the results and state of the search, and attrib is a combination of the following:

`FA_RDONLY`
: Include read-only files in the search (Ignored.)

`FA_HIDDEN`
: Include hidden files in the search

`FA_SYSTEM`
: Include system files in the search

`FA_LABEL`
: Include the volume label in the search

`FA_DIREC`
: Include subdirectories in the search

`FA_ARCH`
: Include modified files in the search (Ignored.)

If a file has flag bits that are not specified in the attrib parameter, the file will be excluded from the results. Thus, if you specified `FA_DIREC` and `FA_LABEL`, subdirectories and the volume label will be included in the results. Hidden and system files will be excluded.

Since `findfirst` calls DOS function 4eh, it is not possible to exclude read-only files or archive files from the results. Even if the FA_ARCH and FA_RDONLY bits are not specified in the attrib parameter, the results will include any read-only and archive files in the directory searched.

This function supports long file names.

The results of the search are stored in ffblk, which is extended when the LFN API (See _use_lfn, LFN) is supported. Fields marked LFN are only valid if the `lfn_magic` member is set to "LFN32".

```
struct ffblk {
char lfn_magic[6]; /* LFN: the magic "LFN32" signature */
short lfn_handle; /* LFN: the handle used by findfirst/findnext */
unsigned short lfn_ctime; /* LFN: file creation time */
unsigned short lfn_cdate; /* LFN: file creation date */
unsigned short lfn_atime; /* LFN: file last access time (usually 0) */
unsigned short lfn_adate; /* LFN: file last access date */
char ff_reserved[5]; /* used to hold the state of the search */
unsigned char ff_attrib; /* actual attributes of the file found */
unsigned short ff_ftime; /* hours:5, minutes:6, (seconds/2):5 */
unsigned short ff_fdate; /* (year-1980):7, month:4, day:5 */
unsigned long ff_fsize; /* size of file */
char ff_name[260]; /* name of file as ASCIIZ string */
}
```

Return Value

Zero if a match is found, nonzero if none found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct ffblk f;
int done = findfirst("*.exe", &f, FA_HIDDEN | FA_SYSTEM);
while (!done)
{

printf("%10u %2u:%02u:%02u %2u/%02u/%4u %s\n",
f.ff_fsize,
(f.ff_ftime >> 11) & 0x1f,
(f.ff_ftime >> 5) & 0x3f,
(f.ff_ftime & 0x1f) * 2,
(f.ff_fdate >> 5) & 0x0f,
(f.ff_fdate & 0x1f),
((f.ff_fdate >> 9) & 0x7f) + 1980,
f.ff_name);
done = findnext(&f);
}
```

findnext

Syntax

```
#include <dir.h>
```

```
int findnext(struct ffblk *ffblk);
```

Description

This finds the next file in the search started by `findfirst`. See findfirst.

Return Value

Zero if there was a match, else nonzero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_fixpath

Syntax

```
#include <sys/stat.h>

void _fixpath(const char *in_path, char *out_path);
```

Description

This function canonicalizes the input path in_path and stores the result in the buffer pointed to by out_path.

The path is fixed by removing consecutive and trailing slashes, making the path absolute if it's relative by prepending the current drive letter and working directory, removing "." components, collapsing ".." components, adding a drive specifier if needed, and converting all slashes to '/'. DOS-style 8+3 names of directories which are part of the pathname, as well as its final filename part, are returned lower-cased in out_path, but long filenames are left intact. See _preserve_fncase, for more details on letter-case conversions in filenames.

Since the returned path name can be longer than the original one, the caller should ensure there is enough space in the buffer pointed to by out_path. Using ANSI-standard constant `FILENAME_MAX` (defined on `stdio.h`) or Posix-standard constant `PATH_MAX` (defined on `limits.h`) is recommended.

Return Value

None. If the length of the returned path name exceeds `FILENAME_MAX`, `errno` is set to `ENAMETOOLONG`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char oldpath[100], newpath[FILENAME_MAX];
scanf("%s", oldpath);
_fixpath(oldpath, newpath);
printf("that really is %s\n", newpath);
```

flock

Syntax

```
#include <sys/file.h>

int flock (int _fildes, int _op);
```

Description

Apply or remove an advisory lock on an open file. The file is specified by file handle _fildes. Valid operations are given below:

`LOCK_SH`
: Shared lock. More than one process may hold a shared lock for a given file at a given time. However, all locks on DOS/Windows 9X are exclusive locks, so `LOCK_SH` requests are treated as if they were `LOCK_EX` requests.

`LOCK_EX`

Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

`LOCK_UN`
Unlock the file.

`LOCK_NB`
Don't block when locking. May be specified (by or'ing) along with one of the other operations.

On other systems, a single file may not simultaneously have both shared and exclusive locks. However, on DOS/Windws 9X, all locks are exclusive locks, so this rule is not true for DOS/Windows 9X.

A file is locked, not the file descriptor. So, `dup (2);` does not create multiple instances of a lock.

Dos/Windows 9X do not support shared locks, but the underlying implementation (which uses the `F_SETLK` (non-blocking) or `F_SETLKW` (blocking) commands to `fcntl`, See fcntl) translates all shared lock request into exclusive lock requests. Thus, requests for shared locks will be treated as if exclusive locks were requested, and only one lock will ever be permitted at any one time on any specified region of the file.

It is therefore wise to code `flock` by `or`ing `LOCK_NB` with all lock requests, whether shared or exclusive, and to test the return value to determine if the lock was obtained or not. Using `LOCK_NB` will cause the implementation to use `F_SETLK` instead of `F_SETLKW`, which will return an error if the lock cannot be obtained.

Return Value

On success, zero is returned. On error, -1 is returned, and `errno` is set appropriately.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. 4.4BSD (the flock (2) call first appeared in 4.2BSD).

Example

```
/* Request a shared lock on file handle fd */
errno = 0;
retval = flock(fd, LOCK_SH);

/* Request a non-blocking shared lock on file handle fd */
errno = 0;
retval = flock(fd, LOCK_SH | LOCK_NB);

/* Request an exclusive lock on file handle fd */
errno = 0;
retval = flock(fd, LOCK_EX);

/* Request a non-blocking exclusive lock on file handle fd */
errno = 0;
retval = flock(fd, LOCK_EX | LOCK_NB);

/* Release a lock on file handle fd */
errno = 0;
retval = flock(fd, LOCK_UN);
```

floor

Syntax

```
#include <math.h>

double floor(double x);
```

Description

This function computes the largest integer not greater than x.

Return Value

The largest integer value less than or equal to x. Infinities and NaNs are returned unchanged.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

_flush_disk_cache

Syntax

```
#include <io.h>

void _flush_disk_cache (void);
```

Description

Attempts to update the disk with the data cached in the write-behind disk caches (such as SmartDrv and the built-in Windows 95 disk cache).

Note that this does **not** flush the DOS buffers. You need to call fsync (See fsync) or close (See close) to force DOS to commit the file data to the disk; sync (See sync) does that for all open files, and also calls _flush_disk_cache.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Make sure all cached data for a handle FD is actually
written to disk. */
fsync (fd);
_flush_disk_cache ();
```

fmod

Syntax

```
#include <math.h>

double fmod(double x, double y);
```

Description

This function computes the remainder of x/y, which is x - iy for some integer i such that iy < x < (i+1)y.

Return Value

The remainder of x/y. If x is Inf or NaN, the return value is NaN and errno is set to EDOM. If y is zero, the return value is zero (but errno is not changed).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

_fmode

Syntax

```
#include <fcntl.h>

extern int _fmode;
```

Description

This variable may be set to O_TEXT or O_BINARY to specify the mode that newly opened files should be opened in if the open call did not specify. See open. See fopen.

The default value is `O_TEXT`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_fmode = O_BINARY;
```

fnmatch

Syntax

```
#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

Description

This function indicates if string matches the pattern. The pattern may include the following special characters:

`*`
: Matches zero of more characters.

`?`
: Matches exactly one character.

`[...]`
: Matches one character if it's in a range of characters. If the first character is `!`, matches if the character is not in the range. Between the brackets, the range is specified by listing the characters that are in the range, or two characters separated by `-` to indicate all characters in that range. For example, `[a-d]` matches `a`, `b`, `c`, or `d`. If you want to include the literal `-` in the range, make it the first character, like in `[-afz]`.

`\`
: Causes the next character to not be treated as a wildcard. For example, `\*` matches an asterisk. This feature is not available if flags includes `FNM_NOESCAPE`, in which case `\` is treated as a directory separator.

The value of flags is a combination of zero of more of the following:

`FNM_PATHNAME`
: This means that the string should be treated as a pathname, in that the slash characters `/` and `\` in string never match any of the wildcards in pattern.

`FNM_NOESCAPE`
: If this flag is **not** set, the backslash `\` may be used in pattern for quoting special characters. If this flag **is** set, `\` is treated as a directory separator.

`FNM_NOCASE`
: If this flag is set, `fnmatch` matches characters case-insensitively, including in character ranges like `[a-f]`. Note that the case-folding is done by calling `toupper` (See toupper), and thus might be sensitive to the current locale.

`FNM_PERIOD`
: This flag is accepted and ignored in the current implementation. (This is the right thing to do on non-LFN platforms, where leading dots in file names are forbidden.)

  In the Posix specification, if this flag is set, leading dots in file names will not match any wildcards. If `FNM_PATHNAME` is set, a dot after a slash also doesn't match any wildcards.

The DJGPP implementation treats forward slashes and backslashes as equal when `FNM_NOESCAPE` is set, since on DOS/Windows these two characters are both used as directory separators in file names.

Return Value

Zero if the string matches, `FNM_NOMATCH` if it does not. Posix defines an additional `FNM_ERROR` code that's returned in case of an error, but the current implementation never returns it.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. The equal handling of `\` and `/` is DJGPP-specific.

Example

```
if (fnmatch("*.[ch]", filename, FNM_PATHNAME|FNM_NOCASE))
do_source_file(filename);
```

fnmerge

Syntax

```
#include <dir.h>

void fnmerge (char *path, const char *drive, const char *dir,
const char *name, const char *ext);
```

Description

This function constructs a file path from its components drive, dir, name, and ext. If any of these is a `NULL` pointer, it won't be used. Usually, the drive string should include the trailing colon `':'`, the dir string should include the trailing slash `'/'` or backslash `'\'`, and the ext string should include the leading dot `'.'`. However, if any of these isn't present, `fnmerge` will add them.

See fnsplit.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[MAXPATH];
fnmerge(buf, "d:", "/foo/", "data", ".txt");
```

fnsplit

Syntax

```
#include <dir.h>

int fnsplit (const char *path, char *drive, char *dir,
char *name, char *ext);
```

Description

This function decomposes a path into its components. It is smart enough to know that `.` and `..` are directories, and that file names with a leading dot, like `.emacs`, are not all extensions.

The drive, dir, name and ext arguments should all be passed, but some or even all of them might be `NULL` pointers. Those of them which are non-`NULL` should point to buffers which have enough room for the strings they would hold. The constants `MAXDRIVE`, `MAXDIR`, `MAXFILE` and `MAXEXT`, defined on dir.h, define the maximum length of these buffers.

See fnmerge.

Return Value

A flag that indicates which components were found:

`DRIVE`
: The drive letter was found.

`DIRECTORY`
: A directory or subdirectories was found.

`FILENAME`
: A filename was found.

`EXTENSION`
: An extension was found.

`WILDCARDS`
: The path included `*` or `?`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char d[MAXDRIVE], p[MAXDIR], f[MAXFILE], e[MAXEXT];
int which = fnsplit("d:/djgpp/bin/gcc.exe", d, p, f, e);
d = "d:"
p = "/djgpp/bin/"
f = "gcc"
e = ".exe"
```

fopen

Syntax

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

Description

This function opens a stream corresponding to the named filename with the given mode. The mode can be one of the following:

`r`
: Open an existing file for reading.

`w`
: Create a new file (or truncate an existing file) and open it for writing.

`a`
: Open an existing file (or create a new one) for writing. The file pointer is positioned to the end of the file before every write.

Followed by any of these characters:

`b`
: Force the file to be open in binary mode instead of the default mode.

  When called to open the console in binary mode, `fopen` will disable the generation of `SIGINT` when you press **Ctrl-C** (**Ctrl-Break**will still cause `SIGINT`), because many programs that use binary reads from the console will also want to get the `^C` characters. You can use the `__djgpp_set_ctrl_c` library function (See __djgpp_set_ctrl_c) if you want **Ctrl-C** to generate interrupts while console is read in binary mode.

`t`
: Force the file to be open in text mode instead of the default mode.

`+`
: Open the file as with `O_RDWR` so that both reads and writes can be done to the same file.

If the file is open for both reading and writing, you must call `fflush`, `fseek`, or `rewind` before switching from read to write or from write to read.

The open file is set to line buffered if the underlying object is a device (stdin, stdout, etc), or is fully buffered if the underlying object is a disk file (data.c, etc).

If `b` or `t` is not specified in mode, the file type is chosen by the value of `fmode` (See _fmode).

You can open directories using `fopen`, but there is limited support for stream file operations on directories. In particular, they cannot be read from or written to.

If you need to specify the DOS share flags use the `__djgpp_share_flags`. See __djgpp_share_flags.

Return Value

A pointer to the `FILE` object, or `NULL` if there was an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
FILE *f = fopen("foo", "rb+"); /* open existing file for read/write,
* binary mode */
```

fork

Syntax

```
#include <unistd.h>

pid_t fork(void);
```

Description

This function always returns -1 and sets `errno` to ENOMEM, as MS-DOS does not support multiple processes. It exists only to assist in porting Unix programs.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

fpathconf

Syntax

```
#include <unistd.h>

long fpathconf(int fd, int name);
```

Description

Returns configuration information on the filesystem that the open file resides on. See pathconf. If the filesystem cannot be determined from the file handle fd (e.g., for character devices), `fpathconf` will return the info for the current drive.

Return Value

The configuration value; for details, see See pathconf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

fpclassify

Syntax

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

Description

The macro `fpclassify` returns the kind of the floating point value supplied.

Return Value

`FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO` or `FP_UNNORMAL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
float f = 1;
double d = INFINITY;
long double ld = NAN;

if( fpclassify(f) != FP_NORMAL )
{

printf("Something is wrong with the implementation!\n");
}

if( fpclassify(d) != FP_INFINITE )
{

printf("Something is wrong with the implementation!\n");
}

if( fpclassify(ld) != FP_NAN )
{

printf("Something is wrong with the implementation!\n");
}
```

__fpclassifyd

Syntax

```
#include <math.h>

int __fpclassifyd(double);
```

Description

Returns the kind of the floating point value supplied. You should use the type generic macro `fpclassify` (See fpclassify) instead of this function.

Return Value

`FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` or `FP_ZERO`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
if( __fpclassifyd(0.0) != FP_ZERO )
{

printf("Something is wrong with the implementation!\n");
}
```

__fpclassifyf

Syntax

```
#include <math.h>

int __fpclassifyf(float);
```

Description

Returns the kind of the floating point value supplied. You should use the type generic macro `fpclassify` (See fpclassify) instead of this function.

Return Value

`FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL` or `FP_ZERO`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
if( __fpclassifyf(0.0F) != FP_ZERO )
{

printf("Something is wrong with the implementation!\n");
}
```

__fpclassifyld

Syntax

```
#include <math.h>

int __fpclassifyld(long double);
```

Description

Returns the kind of the floating point value supplied. You should use the type generic macro `fpclassify` (See fpclassify) instead of this function.

Return Value

`FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO` or `FP_UNNORMAL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
if( __fpclassifyld(0.0L) != FP_ZERO )
{

printf("Something is wrong with the implementation!\n");
}
```

_fpreset

Syntax

```
#include <float.h>

void _fpreset(void);
```

Description

Resets the FPU completely.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fprintf
Syntax
```
#include <stdio.h>

int fprintf(FILE *file, const char *format, ...);
```

Description
Prints formatted output to the named file. See printf.

Return Value
The number of characters written.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

fpurge
Syntax
```
#include <stdio.h>

int fpurge(FILE *file);
```

Description
If file designates a buffered stream open for writing or for both reading and writing, this function purges the stream's buffer without writing it to disk. Otherwise, it does nothing (so it has no effect on read-only streams such as `stdin`).

Return Value
Zero on success, -1 on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

fputc
Syntax
```
#include <stdio.h>

int fputc(int character, FILE *file);
```

Description
This function writes the given character to the given `file`.

Return Value
The given character [0..255] or `EOF`.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example
```
fputc('\n', stdout);
```

fputs

Syntax

```
#include <stdio.h>

int fputs(const char *string, FILE *file);
```

Description

This function all the characters of string (except the trailing `NULL`) to the given file.

Return Value

A nonnegative number on success, `EOF` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
fputs("Hello\n", stdout);
```

fread

Syntax

```
#include <stdio.h>

size_t fread(void *buffer, size_t size, size_t number, FILE *file);
```

Description

This function reads size*number characters from file to buffer.

Return Value

The number of items of size size read, or less if there was an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int foo[10];
fread(foo, sizeof(int), 10, stdin);
```

free

Syntax

```
#include <stdlib.h>

void free(void *ptr);
```

Description

Returns the allocated memory to the heap (See malloc). If the ptr is `NULL`, `free` does nothing.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char *q = (char *)malloc(20);
free(q);
```

freopen

Syntax

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *file);
```

Description

This function closes file if it was open, then opens a new file like `fopen(filename, mode)` (See fopen) but it reuses file.

This is useful to, for example, associate `stdout` with a new file.

Return Value

The new file, or `NULL` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
freopen("/tmp/stdout.dat", "wb", stdout);
```

frexp

Syntax

```
#include <math.h>

double frexp(double x, int *pexp);
```

Description

This function separates the given value x into a mantissa m in the range `[0.5,1)` and an exponent e, such that m*2^e = x. It returns the value of the mantissa and stores the integer exponent in *pexp.

Return Value

The mantissa. If the value of x is `NaN` or `Inf`, the return value is `NaN`, zero is stored in `*pexp`, and `errno` is set to `EDOM`. If x is zero, *pexp and the return value are also both zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

fscanf

Syntax

```
#include <stdio.h>

int fscanf(FILE *file, const char *format, ...);
```

Description

This function scans formatted text from file and stores it in the variables pointed to by the arguments. See scanf.

Return Value

The number of items successfully scanned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

fseek

Syntax

```
#include <stdio.h>

int fseek(FILE *file, long offset, int mode);
```

Description

This function moves the file pointer for file according to mode:

`SEEK_SET`
: The file pointer is moved to the offset specified.

`SEEK_CUR`
: The file pointer is moved relative to its current position.

`SEEK_END`
: The file pointer is moved to a position offset bytes from the end of the file. The offset is usually nonpositive in this case.

Warning! The ANSI standard only allows values of zero for offset when mode is not `SEEK_SET` and the file has been opened as a text file. Although this restriction is not enforced, beware that there is not a one-to-one correspondence between file characters and text characters under MS-DOS, so some `fseek` operations may not do exactly what you expect.

Also, since `lseek` under DOS does not return an error indication when you try to move the file pointer before the beginning of the file, neither will `fseek`. Portable programs should call `ftell` after `fseek` to get the actual position of the file pointer.

Note that DOS does not mind if you seek before the beginning of the file, like seeking from the end of the file by more than the file's size. Therefore, `lseek` will not return with an error in such cases either.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
fseek(stdin, 12, SEEK_CUR); /* skip 12 bytes */
```

fsetpos

Syntax

```
#include <stdio.h>

int fsetpos(FILE *file, const fpos_t *offset);
```

Description

This function moves the file pointer for file to position offset, as recorded by `fgetpos`.

See fgetpos. See fseek.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

__FSEXT_add_open_handler

Syntax

```
#include <sys/fsext.h>

int __FSEXT_add_open_handler(__FSEXT_Function *_function);
```

Description

This function is part of the See File System Extensions. It is used to add a handler for functions that do not get passed descriptors, such as `_open` and `_creat`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
static int
_my_handler(__FSEXT_Fnumber n, int *rv, va_list args)
{

. . .
}


int main()
{

__FSEXT_add_open_handler(_my_handler);
}
```

__FSEXT_alloc_fd

Syntax

```
#include <sys/fsext.h>

int __FSEXT_alloc_fd(__FSEXT_Function *_function);
```

Description

This function is part of the See File System Extensions. It is used by extensions that fully emulate the I/O functions, and thus don't have a corresponding DOS file handle. Upon the first call, this function opens DOS's `NUL` device, so as to allocate a handle that DOS won't then reuse. Upon subsequent calls, that handle is duplicated by calling the DOS `dup` function; this makes all of the handles use a single entry in the System File Table, and thus be independent of what the `FILES=` parameter of `CONFIG.SYS` says. `__FSEXT_alloc_fd` also assigns the handler function for the handle it returns.

The module is responsible for calling `_close` on the descriptor after setting the handler function to zero in the extended close handler.

Return Value

If successful, a new file descriptor is returned. On error, a negative number is returned and `errno` is set to indicate the error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int socket()
{

int fd = __FSEXT_alloc_fd(socket_handler);
init_socket(fd);
return fd;
}
```

__FSEXT_call_open_handlers

Syntax

```
#include <sys/fsext.h>

int __FSEXT_call_open_handlers(__FSEXT_Fnumber _function_number,
int *rv, va_list _args);
```

Description

This function is part of the See File System Extensions. It is used internally to libc.a to allow extensions to get an opportunity to override the `_open` and `_creat` functions.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__FSEXT_get_data

Syntax

```
#include <sys/fsext.h>

void *__FSEXT_get_data(int _fd);
```

Description

This function is part of the See File System Extensions. It is used to retrieve a descriptor-specific pointer that was previously stored by `__FSEXT_set_data` (See __FSEXT_set_data). The pointer is not otherwise used.

See __FSEXT_set_data, for an example of how this may be used.

Return Value

Returns the stored pointer, or NULL if there was an error (or no pointer had been stored).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__FSEXT_get_function

Syntax

```
#include <sys/fsext.h>

__FSEXT_Function *__FSEXT_get_function(int _fd);
```

This function is part of the See File System Extensions. It is used internal to libc.a to redirect I/O requests to the appropriate extensions.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_read(int fd, void *buf, int len)
```

```
{

__FSEXT_Function *func = __FSEXT_get_function(fd);
if (func)
{
int rv;
if (func(__FSEXT_read, &rv, &fd))
return rv;
}
/* rest of read() */
}
```

__FSEXT_set_data

Syntax

```
#include <sys/fsext.h>

void * __FSEXT_set_data(int _fd, void *_data);
```

Description

This function is part of the See File System Extensions. It is used to store a descriptor-specific pointer that can later be retrieved by `__FSEXT_get_data` (See __FSEXT_get_data). The pointer is not otherwise used.

This is useful when writing an extension that may be handling several open pseudo-files. `__FSEXT_set_data` can be used when creating or opening the file to store a pointer to data about the specific file. Later, when specific operation needs to be done (e.g. read, write, etc.) a pointer to pseudo-file associated with the file descriptor can be fetched with `__FSEXT_get_data`.

Return Value

Returns the pointer you passed it, or NULL if there was an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
typedef struct
{

void* Ptr;
off_t Current_Ofs;
size_t Size;
} _mem_file_t;

int my_fsext(__FSEXT_Fnumber Op, int* RV, va_list Args)
{

const char* Path;
void* Buffer;
size_t Size;
int fd;
_mem_file_t* MPtr;

switch (Op)
{
case __FSEXT_creat:
/* Create a new memory file */

Path = va_list(Args, const char*);

/* Check to see if we should create a new file */
if (strnicmp("/tmp/", Path, 5) != 0) return 0;

/* Allocate some memory to keep info on our fake file */
```

```
MPtr = malloc(sizeof(_mem_file_t));
if (!MPtr) return 0;

memset(MPtr, 0, sizeof(_mem_file_t));

/* Get a file descriptor we can use */
fd = __FSEXT_alloc_fd(my_fsext);
if (fd < 0)
{
free(MPtr);
return 0;
}

/* Now store our note about this file descriptor so we can
* look it up quickly later. */
__FSEXT_set_data(fd, MPtr);

/* Return the file descriptor
*RV = fd;
return 1;

case __FSEXT_read:
/* Read from our memory file. */
fd = va_list(Args, int);
Buffer = va_list(Args, void*);
Size = va_list(Args, size_t);

/* Look up the information about this file */
MPtr = __FSEXT_get_data(fd);
if (!MPtr)
{
*RV = -1;
return 1;
}

if (MPtr->Current_Ofs >= MPtr->Size)
{
*RV = 0;
return 1;
}

if (Size > (MPtr->Size - MPtr->Current_Ofs))
Size = MPtr->Size - MPtr->Current_Ofs;

memcpy(Buffer, (char*) MPtr->Ptr+MPtr->Current_Ofs, Size);
MPtr->Current_Ofs += Size;

*RV = Size;
return 1;
...
}
}
```

__FSEXT_set_function

Syntax

```
#include <sys/fsext.h>

int __FSEXT_set_function(int _fd, __FSEXT_Function *_function);
```

Description

This function is part of the See File System Extensions. It is used to set the handler function for those extensions that use DOS files for I/O. One situation where you might need this is when you must catch output to the terminal and play some tricks with it, like colorize it or redirect it to another device.

Return Value

Zero in case of success, non-zero in case of failure (like if _fd is negative).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <sys/fsext.h>
#include <conio.h>

/* A simple example of a write handler which converts DOS I/O to the
screen into direct writes to video RAM. */
static int
my_screen_write (__FSEXT_Fnumber func, int *retval, va_list rest_args)
{

char *buf, *mybuf;
size_t buflen;
int fd = va_arg (rest_args, int);

if (func != __FSEXT_write || !isatty (fd))
return 0; /* and the usual DOS call will be issued */

buf = va_arg (rest_args, char *);
buflen = va_arg (rest_args, size_t);
mybuf = alloca (buflen + 1);
memcpy (mybuf, buf, buflen);
mybuf[buflen] = '\0';
cputs (mybuf);
*retval = buflen;
return 1; /* meaning that we handled the call */
}


/* Install our handler. The 'attribute constructor' causes this
function to be called by the startup code. */
static void __attribute__((constructor))
install_screen_write_handler (void)
{

__FSEXT_set_function (fileno (stdout), my_screen_write);
}
```

fstat

Syntax

```
#include <sys/stat.h>

int fstat(int file, struct stat *sbuf);
```

Description

This function obtains the status of the open file file and stores it in sbuf. See stat, for the description of members of `struct stat`.

The `st_size` member is an signed 32-bit integer type, so it will overflow on FAT32 volumes for files that are larger than 2GB. Therefore, if your program needs to support large files, you should treat the value of `st_size` as an unsigned value.

For some drives `st_blksize` has a default value, to improve performance. The floppy drives A: and B: default to a block size of 512 bytes. Network drives default to a block size of 4096 bytes.

Some members of `struct stat` are very expensive to compute. If your application is a heavy user of `fstat` and is too slow, you can disable computation of the members your application doesn't need, as described in See _djstat_flags.

Return Value

Zero on success, nonzero on failure (and `errno` set).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
struct stat s;
fstat(fileno(stdin), &s);
if (S_ISREG(s.st_mode))
puts("STDIN is a redirected disk file");
else if (S_ISCHR(s.st_mode))
puts("STDIN is a character device");
```

Bugs

If a file was open in write-only mode, its execute and symlink mode bits might be incorrectly reported as if the file were non-executable or non-symlink. This is because executables and symlinks are only recognized by reading their first few bytes, which cannot be done for files open in write-only mode.

For `fstat` to return correct info, you should make sure that all the data written to the file has been delivered to the operating system, e.g. by calling both `fflush` and `fsync`. Otherwise, the buffering of the library I/O functions and the OS might cause stale info to be returned.

Implementation Notes

Supplying a 100% Unix-compatible `fstat` function under DOS is an implementation nightmare. The following notes describe some of the obscure points specific to `fstat`s behavior in DJGPP.

1. The `drive` for character devices (like `con`, `/dev/null` and others is returned as -1. For drives networked by Novell Netware, it is returned as -2.

2. The starting cluster number of a file serves as its inode number. For files whose starting cluster number is inaccessible (empty files, all files on Windows, files on networked drives, etc.) the `st_inode` field will be invented in a way which guarantees that no two different files will get the same inode number (thus it is unique). This invented inode will also be different from any real cluster number of any local file. However, only for local, non-empty files/directories the inode is guaranteed to be consistent between `stat` and `fstat` function calls. (Note that files on different drives can have identical inode numbers, and thus comparing files for identity should include comparison of the `st_dev` member.)

3. On all versions of Windows except Windows 3.X, the inode number is invented using the file name. `fstat` can probably use the file name that was used to open the file, when generating the inode. This is done such that the same inode will be generated irrespective of the actual path used to open the file (e.g.: `foo.txt`, `./foo.txt`, `../somedir/foo.txt`). If file names cannot be used, `fstat` always returns different inode numbers for any two files open on different handles, even if the same file is open twice on two different handles.

4. The WRITE access mode bit is set only for the user (unless the file is read-only, hidden or system). EXECUTE bit is set for directories, files which can be executed from the DOS prompt (batch files, .com, .dll and .exe executables) or run by `go32-v2.exe`. For files which reside on networked drives under Novell Netware, this can sometimes fail, in which case only the read access bit is set.

5. The variable `_djstat_flags` (See _djstat_flags) controls what hard-to-get fields of `struct stat` are needed by the application.

fstatvfs

Syntax

```
#include <sys/types.h>
#include <sys/statvfs.h>

int fstatvfs (int fd, struct statvfs *sbuf);
```

Description

This function returns information about the ''filesystem'' (FS) containing the file referred to by the file descriptor fd and stores it in sbuf, which has the structure below:

```
struct statvfs {
unsigned long f_bsize; /* FS block size */
unsigned long f_frsize; /* fundamental block size */
fsblkcnt_t f_blocks; /* # of blocks on filesystem */
fsblkcnt_t f_bfree; /* # of free blocks on FS */
fsblkcnt_t f_bavail; /* # of free blocks on FS for
* unprivileged users */
fsfilcnt_t f_files; /* # of file serial numbers */
fsfilcnt_t f_ffree; /* # of free file serial numbers */
fsfilcnt_t f_favail; /* # of free file serial numbers
* for unprivileged users */
unsigned long f_fsid; /* FS identifier */
unsigned long f_flag; /* FS flags: bitwise OR of ST_NOSUID,
* ST_RDONLY */
unsigned long f_namemax; /* Maximum file name length on FS */
};
```

Note that if INT 21h is hooked by a TSR, the total size is limited to approximately 2GB (See statvfs).

Return Value

Zero on success, nonzero on failure (and `errno` set).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

fsync

Syntax

```
#include <unistd.h>

int fsync(int file);
```

Description

Forces all information about the file with the given descriptor to be synchronized with the disk image. Works by calling DOS function 0x68. *Warning*: External disk caches are not flushed by this function.

`fsync` does not support directories.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
fsync(fileno(stdout));
```

ftell

Syntax

```
#include <stdio.h>

long ftell(FILE *file);
```

Description

Returns the current file position for `file`. This is suitable for a future call to `fseek`.

Return Value

The file position, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
long p = ftell(stdout);
```

ftime

Syntax

```
#include <sys/timeb.h>

int ftime(struct timeb *buf);
```

Description

This function stores the current time in the structure buf. The format of `struct timeb` is:

```
struct timeb {
time_t time; /* seconds since 00:00:00 GMT 1/1/1970 */
unsigned short millitm; /* milliseconds */
short timezone; /* difference between GMT and local,
* minutes */
short dstflag; /* set if daylight savings time in affect */
};
```

Return Value

Zero on success, nonzero on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct timeb t;
ftime(&t);
```

ftruncate

Syntax

```
#include <unistd.h>

int ftruncate(int handle, off_t where);
```

Description

This function truncates the file open on handle at byte position where. The file pointer associated with handle is not changed.

Note that this function knows nothing about buffering by stdio functions like `fwrite` and `fprintf`, so if handle comes from a `FILE` object, you need to call `fflush` before calling this function.

`ftruncate` does not support directories.

Return Value

Zero for success, nonzero for failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int x = open("data", O_WRONLY);
ftruncate(x, 1000);
close(x);
```

ftw

Syntax

```
#include <ftw.h>

int ftw(const char *dir,
int (*func)(const char *path, struct stat *stbuf, int flag),
int depth);
```

Description

This function recursively descends the directory hierarchy which starts with dir. For each file in the hierarchy, `ftw` calls the user-defined function func which is passed a pointer to a `NULL`-terminated character array in path holding the full pathname of the file, a pointer to a `stat` structure (See stat) stbuf with a filesystem information about that file, and an integer flag. Possible values of flag are:

`FTW_F`
: This is a regular file.

`FTW_D`
: This is a directory.

`FTW_VL`
: This is a volume label.

`FTW_DNR`
: This is a directory which cannot be read with `readdir()`. (This will never happen in DJGPP.)

`FTW_NS`
: This file exists, but `stat` fails for it.

If flag is `FTW_DNR`, the descendants of that directory won't be processed. If flag is `FTW_NS`, then stbuf will be garbled.

This function always visits a directory before any of its siblings. The argument dir must be a directory, or `ftw` will fail and set `errno` to `ENOTDIR`. The function func is called with dir as its argument before the recursive descent begins.

The depth argument has no meaning in the DJGPP implementation and is always ignored.

The tree traversal continues until one of the following events:

(1) The tree is exhausted (i.e., all descendants of dir are processed). In this case, `ftw` returns 0, meaning a success.

(2) An invocation of func returns a non-zero value. In this case, `ftw` stops the tree traversal and returns whatever func returned.

(3) An error is detected within `ftw`. In that case, `ftw` returns -1 and sets `errno` (See errno) to a suitable value.

Return Value

Zero in case the entire tree was successfully traversed, -1 if `ftw` detected some error during its operation, or any other non-zero value which was returned by the user-defined function func.

Implementation Notes

This function uses `malloc` (See malloc) for dynamic memory allocation during its operation. If func disrupts the normal flow of code execution by e.g. calling `longjump` or if an interrupt handler which never returns is executed, this memory will remain permanently allocated.

This function calls `opendir()` and `readdir()` functions to read the directory entries. Therefore, you can control what files will your func get by setting the appropriate bits in the external variable __opendir_flags. See opendir, for description of these bits.

This function also calls `stat` for every directory entry it passes to func. If your application only needs some part of the information returned in the `stat` structure, you can make your application significantly faster by setting bits in the external variable _djstat_flags (See _djstat_flags for details). The most expensive `stat` features are `_STAT_EXEC_MAGIC` and `_STAT_DIRSIZE`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <stdlib.h>

int
file_walker(const char *path, struct stat *sb, int flag)
{

char *base;

printf("%s:\t%u\t", path, sb->st_size);
if (S_ISLABEL(sb->st_mode))
printf("V");
if (S_ISDIR(sb->st_mode))
printf("D");
if (S_ISCHR(sb->st_mode))
printf("C");
if (sb->st_mode & S_IRUSR)
printf("r");
if (sb->st_mode & S_IWUSR)
printf("w");
if (sb->st_mode & S_IXUSR)
printf("x");

if (flag == FTW_NS)
printf(" !!no_stat!!");
printf("\n");

base = strrchr(path, '/');
if (base == 0)
base = strrchr(path, '\\');
if (base == 0)
base = strrchr(path, ':');
if (strcmp(base == 0 ? path : base + 1, "xxxxx") == 0)
return 42;
return 0;
}


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

if (argc > 1)
{
char msg[80];

sprintf(msg, "file_tree_walk: %d",
ftw(argv[1], file_walker, 0));
if (errno)
perror(msg);
else
puts(msg);
}
else
```

```
printf("Usage: %s dir\n", argv[0]);

return 0;
}
```

_fwalk

Syntax

```
#include <libc/file.h>

void _fwalk(void (*function)(FILE *file));
```

Description

For each open file in the system, the given function is called, passing the file pointer as its only argument.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
void pfile(FILE *x)
{ printf("FILE at %p\n", x); }

_fwalk(pfile);
```

fwrite

Syntax

```
#include <stdio.h>

size_t fwrite(void *buffer, size_t size, size_t number, FILE *file);
```

Description

This function writes size*number characters from buffer to file.

Return Value

The number of items of size size written, or less if there was an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int foo[10];
fwrite(foo, sizeof(int), 10, stdin);
```

gcvt

Syntax

```
#include <stdlib.h>

char * gcvt (double value, int ndigits, char *buf)
```

Description

This function converts its argument value into a null-terminated string of ndigits significant digits in buf. buf should have enough space to hold at least `ndigits + 7` characters. The result roughly corresponds to what is obtained by the following snippet:

```
(void) sprintf(buf, "%.*g", ndigits, value);
```

except that trailing zeros and trailing decimal point are suppressed.

The least-significant digit in buf is rounded.

`ecvtbuf` produces the string ‘‘NaN’’ if value is a NaN, and ‘‘Inf’’ if value is an infinity.

Return Value

A pointer to buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

char vbuf[20];

/* This will print " 3.14159". */
printf ("%s", gcvt (M_PI, 5, buf));
```

_get_dev_info

Syntax

```
#include <io.h>

short _get_dev_info(int handle);
```

Description

Given a file handle in handle, this function returns the info word from DOS IOCTL function 0 (Int 21h/AX=4400h). handle must refer to an open file or device, otherwise the call will fail (and set `errno` to `EBADF`).

In case of success, the returned value is the coded information from the system about the character device or the file which is referenced by the file handle handle. The header `<libc/getdinfo.h>` defines constants for the individual bits in the return value. The following table shows the meaning of the individual bits in the return value:

For a character device:

{Bit(s) {`_DEV_STDOUT` {Device can process IOCTL functions 02h and 03h Constant Description

`_DEV_IOCTRL` Device can process IOCTL functions 02h and 03h

Device supports output-until-busy

Device supports OPEN/CLOSE calls

Unknown; set by MS-DOS 6.2x `KEYBxx.COM`

`_DEV_CDEV` Always set for character devices

End of file on input

`_DEV_RAW` If set, device is in **raw** (binary) mode

`_DEV_RAW` If clear, device is in **cooked** (text) mode

Device uses Int 29h

`_DEV_CLOCK` Clock device

`_DEV_NUL` NUL device

_DEV_STDOUT Standard output device

_DEV_STDIN Standard input device

For a block device (a disk file):

{Bit(s) {_DEV_STDOUT {Generate Int 24h if full disk or read past EOF Constant Description

_DEV_REMOTE Device is remote (networked drive)

Don't set file time stamp on close

If set, non-removable media

If clear, media is removable (e.g. floppy disk)

Generate Int 24h if full disk or read past EOF

_DEV_CDEV Always clear for disk files

File has not been written to

Drive number (0 = A:)

Note that the functionality indicated by bit 8 for the block devices is only supported by DOS version 4.

Cooked mode means that on input **C-C**, **C-P**, **C-S** and **C-Z** are processed, on output TABs are expanded into spaces and CR character is added before each LF, and input is terminated when the **RET** key is pressed. In contrast, in raw mode, all the special characters are passed verbatim, and the read operation waits until the specified number of characters has been read.

Return Value

The device information word described above. In case of error, -1 is returned and errno is set to EBADF.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int fd = open ("CLOCK$", O_RDONLY | O_BINARY);
int clock_info = _get_dev_info (fd);
```

_get_dos_version

Syntax

```
#include <dos.h>

extern unsigned short _osmajor, _osminor;
extern unsigned short _os_trueversion;
extern const char * _os_flavor;

unsigned short _get_dos_version(int true_version);
```

Description

This function gets the host OS version and flavor. If the argument true_version is non-zero, it will return a true version number, which is unaffected by possible tinkering with SETVER TSR program. (This is only available in DOS 5.0 or later.)

The external variables _osmajor and _osminor will always be set to the major and minor parts of the advertised version number. The external variable _os_trueversion will always be set to the true version number. _osmajor, _osminor and _os_trueversion may possibly be changed by SETVER, even if true_version is non-zero.

You typically need the true version when you need an intimate knowledge of the host OS internals, like when using undocumented features. Note that some DOS clones (notably, DR-DOS) do not support DOS function required to report the true DOS version; for these, the version reported might be affected by SETVER even if true_version is non-zero.

The external variable `_os_flavor` will point to a string which describes the OEM name of the host OS variety.

Return Value

`_get_dos_version()` returns the version number (true version number, if true_version is non-zero) as a 16-bit number: the major part of the version in the upper 8 bits, the minor part in the lower 8 bits. For instance, DOS version 6.20 will be returned as 0x0614.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned short true_dos_version = _get_dos_version(1);

if (true_dos_version < 0x0614) /* require DOS 6.20 or later */
puts("This program needs DOS 6.20 or later to run");
else
printf("You are running %s variety of DOS\n", _os_flavor);
```

__get_extended_key_string

Syntax

```
#include <pc.h>

const unsigned char * __get_extended_key_string(int xkey_code);
```

Description

Returns an ECMA-48 compliant representation of an extended key's scan code in xkey_code.

See getkey. See getxkey.

Return Value

A string based on the extended key's scan code xkey_code:

{**Page Down** {ESC[24~ {with Shift {with Ctrl {ESC[80~

ESC[A ESC[37~ ESC[59~

ESC[B ESC[38~ ESC[60~

ESC[C ESC[39~ ESC[61~

ESC[D ESC[40~ ESC[62~

ESC[1~ ESC[41~ ESC[63~

ESC[2~ ESC[42~ ESC[64~

ESC[3~ ESC[43~ ESC[65~

ESC[4~ ESC[44~ ESC[66~

ESC[5~ ESC[45~ ESC[67~

ESC[6~ ESC[46~ ESC[68~

ESC[[A ESC[25~ ESC[47~ ESC[69~

ESC[[B ESC[26~ ESC[48~ ESC[70~

ESC[[C ESC[27~ ESC[49~ ESC[71~

ESC[[D ESC[28~ ESC[50~ ESC[72~

ESC[[E ESC[29~ ESC[51~ ESC[73~

ESC[17~ ESC[30~ ESC[52~ ESC[74~

ESC[18~ ESC[31~ ESC[53~ ESC[75~

ESC[19~ ESC[32~ ESC[54~ ESC[76~

ESC[20~ ESC[33~ ESC[55~ ESC[77~

ESC[21~ ESC[34~ ESC[56~ ESC[78~

ESC[23~ ESC[35~ ESC[57~ ESC[79~

ESC[24~ ESC[36~ ESC[58~ ESC[80~

{**Alt-A** {ESC[83~ {Alt-Z {ESC[113~ **Alt-N** ESC[94~ **Alt-O** ESC[95~ **Alt-P** ESC[96~ **Alt-Q** ESC[97~ **Alt-R** ESC[98~ **Alt-S** ESC[99~ **Alt-T** ESC[100~ **Alt-U** ESC[101~ **Alt-V** ESC[102~ **Alt-W** ESC[103~ **Alt-X** ESC[104~ **Alt-Y** ESC[105~ **Alt-Z** ESC[106~

NULL is returned if xkey_code has no translation.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int key;

int main()
{

key = getxkey();
if (key < 0x100)
{
putc(key, stdout);
putc('\r', stdout);
}
else
{
const unsigned char *str = __get_extended_key_string(key);
if (str)
puts(str);
else
puts("<unknown>");
}
fflush(stdout);
}



#include <pc.h>
#include <stdio.h>
#include <dpmi.h>

int main()
{

__dpmi_regs r;
const unsigned char *str;
int is_extended_key;
```

```
/* Wait for keypress. */
r.h.ah = 0x11;
__dpmi_int(0x16, &r);
/* Print the encoding for function keys (F1, F2, etc.)
and other extended keys (Home, End, etc.). */
is_extended_key = (r.h.al == 0x00 || r.h.al == 0xe0);
if (is_extended_key)
{
str = __get_extended_key_string((int)r.h.ah)
printf("Key encoding: %s", str);
}
}
```

_get_fat_size

Syntax

```
#include <dos.h>

int _get_fat_size( const int drive );
```

Description

This function tries to determine the number of bits used to enumerate the clusters by the FAT on drive number drive. 1 == A:, 2 == B:, etc., 0 == default drive.

This function will not succeed on DOS version < 4, setting `errno` to `ENOSYS`. It is also known to have trouble detecting the file system type of disks formatted with a later version of DOS than the version it is run on. E. g. floppies with LFN entries can cause this function to fail or detect the fat size as 16 if used in plain DOS.

If you are looking for a function that is able to detect other file systems, perhaps the function `_get_fs_type` (See _get_fs_type) can be of use.

Return Value

The number of bits used by the FAT (12, 16 or 32). 0 if the drive was formatted with FAT but of unknown size (NT reports that on FAT16). -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{

int size;

size = _get_fat_size( 'C' - 'A' + 1 );
if( 0 <= size )
{
printf("The size of FAT on C: is %d bits.\n", size);
}

exit(0);
}
```

__get_fd_flags

Syntax

```
#include <libc/fd_props.h>
```

```
unsigned long __get_fd_flags(int fd);
```

Description

This internal functions gets the flags associated with file descriptor fd, if any. The flags are some properties that may be associated with a file descriptor (See __set_fd_properties).

This function will return zero, if no flags are associated with fd. The caller should first check that there are flags associated with fd using `__has_fd_properties` (See __has_fd_properties), before calling `__get_fd_flags`. This will allow the cases of no flags and the flags being zero to be distinguished.

Return Value

The flags, if any; otherwise zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__get_fd_name

Syntax

```
#include <libc/fd_props.h>

const char *__get_fd_name(int fd);
```

Description

This internal function gets the file name associated with the file descriptor fd, if any. The file name is one property that may be associated with a file descriptor (See __set_fd_properties).

Return Value

A pointer to the file name, if any; otherwise `NULL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_get_fs_type

Syntax

```
#include <dos.h>

int _get_fs_type( const int drive,
char *const result_str );
```

Description

This function tries to extract the file system type of the drive number drive, 1 == A:, 2 == B:, etc., 0 == default drive. It does this by calling INT21, AX=0x6900, Get Disk Serial Number (sic!), which returns, among other things, an eight character field which is set while formatting the drive. Now, this field can be set to whatever the formatting program wishes, but so far every FAT formatted drive has returned a string starting with "FAT".

If successful the result is put in result_str which must be at least 9 characters long. If unsuccessful `errno` is set.

This function will not succeed on DOS version < 4, setting `errno` to `ENOSYS`. It is also known to have trouble detecting the file system type of disks formatted with a later version of DOS than the version it is run on. E. g. floppies with LFN entries can cause this function to fail or detect the floppy as FAT16 if used in plain DOS.

If you are interested in which kind of FAT file system that is in use try the function `_get_fat_size` (See _get_fat_size) which will reliably detect the right kind of FAT file system.

Return Value

0 if the file system type was extracted successfully; otherwise -1.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{

char buffer[9];

if( ! _get_fs_type( 3, buffer ) )
{
printf("The file system on C: is ’%s’.\n", buffer);
}

exit(0);
}
```

_get_volume_info

Syntax

```
#include <fcntl.h>
unsigned _get_volume_info (const char *path,
int *max_file_len, int *max_path_len,
char *fsystype);
```

Description

This function returns filesystem information about the volume where path resides. Only the root directory name part is actually used; if path does not specify the drive explicitly, or is a `NULL` pointer, the current drive is used. Upon return, the variable pointed to by max_file_len contains the maximum length of a filename (including the terminating zero), the variable pointed to by max_path_len contains the maximum length of a pathname (including the terminating zero), and a string that identifies the filesystem type (e.g., ‘‘FAT’’, ‘‘NTFS’’ etc.) is placed into the buffer pointed to by fsystype, which should be long enough (32 bytes are usually enough). If any of these pointers is a `NULL` pointer, it will be ignored. The function returns various flags that describe features supported by the given filesystem as a bit-mapped number. The following bits are currently defined:

`_FILESYS_CASE_SENSITIVE`
: Specifies that file searches are case-sensitive.

`_FILESYS_CASE_PRESERVED`
: Filename letter-case is preserved in directory entries.

`_FILESYS_UNICODE`
: Filesystem uses Unicode characters in file and directory names.

`_FILESYS_LFN_SUPPORTED`
: Filesystem supports the Long File Name (LFN) API.

`_FILESYS_VOL_COMPRESSED`
: This volume is compressed.

`_FILESYS_UNKNOWN`
: The underlying system call failed. This usually means that the drive letter is invalid, like when a floppy drive is empty or a drive with that letter doesn’t exist.

Return value

A combination of the above bits if the LFN API is supported, or 0 (and `errno` set to `ENOSYS`) if the LFN API is not supported by the OS. If the LFN API is supported, but the drive letter is invalid, the function returns `_FILESYS_UNKNOWN` and sets `errno` to either `ENODEV` or `ENXIO`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getc

Syntax

```
#include <stdio.h>

int getc(FILE *file);
```

Description

Get one character from file.

Return Value

The character ([0..255]) or `EOF` if eof or error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int c;
while ((c=getc(stdin)) != EOF)
putc(c, stdout);
```

getcbrk

Syntax

```
#include <dos.h>

int getcbrk(void);
```

Description

Get the setting of the Ctrl-C checking flag in MS-DOS.

See setcbrk.

Return Value

0 if not checking, 1 if checking.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getch

Syntax

```
#include <conio.h>

int getch(void);
```

Description

A single character from the predefined standard input handle is read and returned. The input is not buffered. If there is a character pending from `ungetch` (See ungetch), it is returned instead. The character is not echoed to the screen. This function doesn't check for special characters like **Ctrl-C**.

If the standard input handle is connected to the console, any pending output in the `stdout` and `stderr` streams is flushed before reading the input, if these streams are connected to the console.

Return Value

The character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

getchar

Syntax

```
#include <stdio.h>

int getchar(void);
```

Description

The same as `fgetc(stdin)` (See fgetc).

Return Value

The character, or `EOF`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

getche

Syntax

```
#include <conio.h>

int getche(void);
```

Description

A single character from the predefined standard input handle is read and returned. The input is not buffered. If there is a character pending from `ungetch` (See ungetch), it is returned instead. The character is echoed to the screen. This function doesn't check for special characters like **Ctrl-C**.

If the standard input handle is connected to the console, any pending output in the `stdout` and `stderr` streams is flushed before reading the input, if these streams are connected to the console.

Return Value

The character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

getcwd

Syntax

```
#include <unistd.h>

char *getcwd(char *buffer, int max);
```

Description

Get the current directory. The return value includes the drive specifier.

If buffer is `NULL`, `getcwd` allocates a buffer of size max with `malloc`. This is an extension of the POSIX standard, which is compatible with the behaviour of glibc (the C library used on Linux).

This call fails if more than max characters are required to specify the current directory.

Return Value

The buffer, either buffer or a newly-allocated buffer, or `NULL` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. The behaviour when buffer is `NULL` is unspecified for POSIX.

Example

```
char *buf = (char *)malloc(PATH_MAX);
if (buf && getcwd(buf, PATH_MAX))
{

printf("cwd is %s\n", buf);
free(buf);
}
```

getdate

Syntax

```
#include <dos.h>

void getdate(struct date *);
```

Description

This function gets the current date. The return structure is as follows:

```
struct date {
short da_year;
char da_day;
char da_mon;
};
```

See setdate. See gettime.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct date d;
getdate(&d);
```

getdfree

Syntax

```
#include <dos.h>

void getdfree(unsigned char drive, struct dfree *ptr);
```

Description

This function gets information about the size and fullness of the given drive (0=default, 1=A:, etc). The return structure is as follows:

```
struct dfree {
unsigned df_avail; /* number of available clusters */
unsigned df_total; /* total number of clusters */
unsigned df_bsec; /* bytes per sector */
unsigned df_sclus; /* sectors per cluster */
};
```

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct dfree d;
getdfree(3, &d); /* drive C: */
```

getdisk

Syntax

```
#include <dir.h>

int getdisk(void);
```

Description

Gets the current disk (0=A).

See setdisk.

Return Value

The current disk number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("This drive is %c:\n", getdisk() + 'A');
```

getdtablesize

Syntax

```
#include <unistd.h>

int getdtablesize(void);
```

Description

Get the maximum number of open file descriptors the system supports.

Return Value

255

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getegid

Syntax

```
#include <unistd.h>

int getegid(void);
```

Description

Get the effective group id.

Return Value

42

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getenv

Syntax

```
#include <stdlib.h>

char *getenv(const char *name);
```

Description

Get the setting of the environment variable name. Do not alter or free the returned value.

Return Value

The value, or `NULL` if that variable does not exist.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char *term = getenv("TERM");
```

geteuid

Syntax

```
#include <unistd.h>

int geteuid(void);
```

Description

Gets the effective UID.

Return Value

42

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getftime

Syntax

```
#include <dos.h>

int getftime(int handle, struct ftime *ptr);
```

Description

Get the timestamp for the given file handle. The return structure is as follows:

```
struct ftime {
unsigned ft_tsec:5; /* 0-29, double to get real seconds */
unsigned ft_min:6; /* 0-59 */
unsigned ft_hour:5; /* 0-23 */
unsigned ft_day:5; /* 1-31 */
unsigned ft_month:4; /* 1-12 */
unsigned ft_year:7; /* since 1980 */
}
```

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct ftime t;
getftime(fd, &t);
```

getgid

Syntax

```
#include <unistd.h>

int getgid(void);
```

Description

Get the current group id.

Return Value

42

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getgrent

Syntax

```
#include <grp.h>

struct group *getgrent(void);
```

Description

This function returns the next available group entry. Note that for MS-DOS, this is simulated. If the environment variable `GROUP` is set, that is the name of the only group returned, else the only group is "dos". Thus, under DOS, `getgrent` will always fail on the second and subsequent calls.

The return type of this and related function is as follows:

```
struct group {
gid_t gr_gid; /* result of getgid() */
char ** gr_mem; /* gr_mem[0] points to
getenv("USER"/"LOGNAME") or "user" */
char * gr_name; /* getenv("GROUP") or "dos" */
char * gr_passwd; /* "" */
};
```

Return Value

The next structure, or `NULL` at the end of the list.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct group *g;
setgrent();
while ((g = getgrent()) != NULL)
{

printf("group %s gid %d\n", g->gr_name, g->gr_gid);
}

endgrent();
```

getgrgid

Syntax

```
#include <grp.h>

extern struct group *getgrgid(int gid);
```

Description

This function returns the group entry that matches gid. See getgrent, for the description of `struct group`.

Return Value

The matching group, or `NULL` if none match.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getgrnam

Syntax

```
#include <grp.h>

struct group *getgrnam(char *name);
```

Description

This function returns the group entry for the group named name. See getgrent, for the description of `struct group`.

Return Value

The matching group, or `NULL` if none match.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getgroups

Syntax

```
#include <unistd.h>

int getgroups(int size, gid_t *grouplist);
```

Description

This function always returns zero. It exists to assist porting from Unix.

Return Value

Zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

gethostname

Syntax

```
#include <unistd.h>
#include <sys/param.h>

int gethostname (char *buf, int size);
```

Description

Get the name of the host the program is executing on. This name is obtained from the network software, if present, otherwise from the `"HOSTNAME"` environment variable, if present, finally defaulting to `"pc"`.

The call fails if more than size characters are required to specify the host name. A buffer size of `MAXGETHOSTNAME` is guaranteed to be enough.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *buf = (char *) malloc (MAXGETHOSTNAME);
if (buf && 0 == gethostname (buf, MAXGETHOSTNAME))
printf ("We're on %s\n", buf);
if (buf) free(buf);
```

getitimer

Syntax

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
```

Description

This function gets the current value of the interval timer specified by which into structure value. Variable which can have the value of `ITIMER_REAL` or `ITIMER_PROF`. See setitimer, for more details about timers.

Upon return, the `it_value` member of value will hold the amount of time left until timer expiration, or zero if the timer has expired or was stopped by a previous call to `setitimer`. The `it_interval` member will hold the interval between two successive alarms as set by the last call to `setitimer` (but note that interval values less than the system clock granularity are rounded up to that granularity). The value returned in `it_interval` member is *not* set to zero when the timer is stopped, it always retains the interval that was last in use.

Return Value

Returns 0 on success, -1 on failure (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getkey

Syntax

```
#include <pc.h>
#include <keys.h>

int getkey(void);
```

Description

Waits for the user to press one key, then returns that key. Alt-key combinations have 0x100 added to them. Extended keys return their non-extended codes.

The file `keys.h` has symbolic names for many of the keys.

See getxkey.

Return Value

The key pressed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (getkey() != K_Alt_3)
do_something();
```

getlogin

Syntax

```
#include <unistd.h>

char *getlogin(void);
```

Description

Get the login ID of the user.

Return Value

Returns the value of the `USERNAME` environment variable if it is defined, else the `LOGNAME` environment variable, else the `USER` environment variable, else `"dosuser"`.

`USERNAME` is set automatically by Windows NT and Windows 2000. None of these environment variables are set automatically on DOS, Windows 95 or Windows 98.

The stock version of the file `DJGPP.ENV` defines `USER` with the value `"dosuser"`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
printf("I am %s\n", getlogin());
```

getlongpass

Syntax

```
#include <stdlib.h>

int getlongpass(const char *prompt, char *password, int max_length)
```

Description

This function reads up to a Newline (CR or LF) or EOF (Ctrl-D or Ctrl-Z) from the standard input, without an echo, after prompting with a null-terminated string prompt. It puts a null-terminated string of at most max_length - 1 first characters typed by the user into a buffer pointed to by password. Pressing Ctrl-C or Ctrl-Break will cause the calling program to `exit(1)`.

Return Value

Zero if successful, -1 on error (and `errno` is set to an appropriate value).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char password[MAX_PASS];

(void)getlongpass("Password: ", password, MAX_PASS);
```

getmntent

Syntax

```
#include <mntent.h>

struct mntent *getmntent(FILE *filep);
```

Description

This function returns information about the various drives that are available to your program. Beginning with drive `A:`, information is retrieved for successive drives with successive calls to `getmntent`. Note that drives `A:` and `B:` will only be returned if there is an MS-DOS formatted disk in the drive; empty drives are skipped. For systems with a single floppy drive, it is returned as if it were mounted on `A:/` or `B:/`, depending on how it was last referenced (and if there is a disk in the drive).

The argument filep should be a `FILE*` pointer returned by `setmntent` (See setmntent).

For each drive scanned, a pointer to a static structure of the following type is returned:

```
struct mntent
{

char * mnt_fsname; /* The name of this file system */
char * mnt_dir; /* The root directory of this file system */
char * mnt_type; /* Filesystem type */
char * mnt_opts; /* Options, see below */
int mnt_freq; /* -1 */
int mnt_passno; /* -1 */
long mnt_time; /* -1 */
};
```

DJGPP implementation returns the following in the first 4 fields of `struct mntent`:

`mnt_fsname`
: For networked and CD-ROM drives, this is the name of root directory in the form `\\HOST\PATH` (this is called a UNC name).

For drives compressed with DoubleSpace, `mnt_fsname` is the string `X:\DBLSPACE.NNN`, where X is the drive letter of the host drive and NNN is the sequence number of the Compressed Volume File.

For drives compressed with Stacker, `mnt_fsname` is the string `X:\STACVOL.NNN`, where X and NNN are as for DoubleSpace drives.

For drives compressed with Jam (a shareware disk compression software), `mnt_fsname` is the full name of the Jam archive file.

For SUBSTed drives, `mnt_fsname` is the actual directory name that that was SUBSTed to emulate a drive.

JOINed drives get their `mnt_fsname` as if they were NOT JOINed (i.e., either the label name or the default `Drive X:`).

For drives with a volume label, `mnt_fsname` is the name of the label; otherwise the string `Drive X:`, where X is the drive letter.

`mnt_dir`

For most drives, this is the name of its root directory `X:/` (where X is the drive letter), except that JOINed drives get `mnt_dir` as the name of the directory to which they were JOINed.

For systems with a single floppy drive (which can be referenced as either `a:/` or `b:/`), the mount directory will be returned as one of these, depending on which drive letter was last used to reference that drive.

`mnt_type`

```
"fd" for floppy disks
"hd" for hard disks
"dblsp" for disks compressed with DoubleSpace
"stac" for disks compressed with Stacker
"jam" for disks compressed with Jam
"cdrom" for CD-ROM drives
"ram" for RAM disks
"subst" for SUBSTed directories
"join" for JOINed disks
"net" for networked drives
```

`mnt_opts`

The string `ro,dev=XX` for CD-ROM drives, `rw,dev=XX` for all the others, where XX is the hexadecimal drive number of the REAL drive on which this filesystem resides. That is, if you call `stat` on mnt_fsname, you will get the numeric equivalent of XX in `st_dev` member of `struct stat`. E.g., for drive `C:` you will get `rw,dev=02`. Note that SUBSTed and JOINed drives get the drive numbers as if SUBST and JOIN were **not** in effect.

Return Value

This function returns a pointer to a `struct mntent`, or `NULL` if there are no more drives to report on.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct mntent *m;
FILE *f;
f = setmntent("/etc/mnttab", "r");
while ((m = getmntent(f)))
printf("Drive %s, name %s\n", m->mnt_dir, m->mnt_fsname);
endmntent(f);
```

getopt

Syntax

```
#include <unistd.h>

int getopt(int argc, char * const *argv, const char *options);
extern char *optarg;
extern int optind, opterr, optopt;
```

Description

Parse options from the command line. options is a string of valid option characters. If a given option takes an argument, that character should be followed by a colon.

For each valid switch, this function sets `optarg` to the argument (if the switch takes one), sets `optind` to the index in argv that it is using, sets `optopt` to the option letter found, and returns the option letter found.

If an unexpected option is found, a question mark (`?`) is returned. If an option argument is missing, a colon (`:`) is returned if the first character in options is a colon, otherwise a question mark is returned. In both cases, if `opterr` is nonzero and the first character in options is not a colon, an error message is printed to `stderr`.

The example below shows how `?` can still be used as an option (e.g., to request a summary of program usage) in addition to flagging an unexpected option and a missing argument.

Return Value

The option character is returned when found. -1 is returned when there are no more options. A colon (`:`) is returned when an option argument is missing and the first character in options is a colon. A question mark (`?`) is returned when an invalid option is found or an option argument is missing and the first character in options is not a colon.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int c;
opterr = 0;
while ((c=getopt(argc, argv, ":?vbf:")) != -1)
{

switch (c)
{
case 'v':
verbose_flag ++;
break;
case 'b':
binary_flag ++;
break;
case 'f':
output_filename = optarg;
break;
case ':':
printf("Missing argument %c\n", optopt);
usage();
exit(1);
case '?':
if (optopt == '?') {
usage();
exit(0);
} else {
printf("Unknown option %c\n", optopt);
usage();
exit(1);
}
}
}
```

getpagesize

Syntax

```
#include <unistd.h>

int getpagesize(void);
```

Description

Return the size of the native virtual memory page size.

Return Value

4096 for the i386 and higher processors.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getpass

Syntax

```
#include <stdlib.h>

char * getpass(const char *prompt)
```

Description

This function reads up to a Newline (CR or LF) or EOF (Ctrl-D or Ctrl-Z) from the standard input, without an echo, after prompting with a null-terminated string prompt. It returns the string of at most 8 characters typed by the user. Pressing Ctrl-C or Ctrl-Break will cause the calling program to `exit(1)`.

Return Value

A pointer to a static buffer which holds the user's response. The buffer will be overwritten by each new call. In case of any error in the lower I/O routines, a NULL pointer will be returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *password = getpass("Password: ");
```

getpgrp

Syntax

```
#include <unistd.h>

int getpgrp(void);
```

Description

Gets the process group, which is currently the same as the pid.

Return Value

The process group.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getpid

Syntax

```
#include <unistd.h>

int getpid(void);
```

Description

Get the process ID, which uniquely identifies each program running on the system.

Return Value

The process ID.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getppid

Syntax

```
#include <unistd.h>

int getppid(void);
```

Description

Get the parent process ID. Currently this is always 1, indicating that the parent process is no longer running.

Return Value

The parent process ID.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getpwent

Syntax

```
#include <pwd.h>

struct passwd *getpwent(void);
```

Description

This function retrieves the next available password file entry. For MS-DOS, this is simulated by providing exactly one entry:

```
struct passwd {
char * pw_name; /* getlogin() */
int pw_uid; /* getuid() */
int pw_gid; /* getgid() */
char * pw_dir; /* "/" or getenv("HOME") */
char * pw_shell; /* "/bin/sh" or getenv("SHELL") */
char * pw_gecos; /* getlogin() */
char * pw_passwd; /* "" */
};
```

The `pw_name` and `pw_gecos` members are returned as described under `getlogin` (See getlogin). The `pw_uid` member is returned as described under `getuid` (See getuid). `pw_gid` is returned as described under `getgid` (See getgid). The `pw_passwd` member is set to the empty string. The `pw_dir` member is set to the value of the environment variable `HOME` if it is defined, or to `/` otherwise. `pw_shell` is set as follows:

- If the environment variable `SHELL` is set, the value of `SHELL`.
- If `SHELL` is not set, but the environment variable `COMSPEC` is, the value of `COMSPEC`.
- If neither of the above variables is defined, `pw_shell` is set to `"sh"`.

Return Value

The next passwd entry, or `NULL` if there are no more.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct passwd *p;
setpwent();
while ((p = getpwent()) != NULL)
{

printf("user %s name %s\n", p->pw_name, p->pw_gecos);
}

endpwent();
```

getpwnam

Syntax

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
```

Description

This function gets the password file entry matching name. See See getpwent, for the description of `struct passwd`.

Return Value

The matching record, or `NULL` if none match.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getpwuid

Syntax

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

Description

This function gets the password file entry matching uid. See See getpwent, for the description of `struct passwd`.

Return Value

The matching record, or `NULL` if none match.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getrlimit

Syntax

```
#include <sys/resource.h>

int getrlimit (int rltype, struct rlimit *rlimitp);
```

Description

This function gets the resource limit specified by rltype and stores it in the buffer pointed to by rlimitp. The `rlimit` structure is defined on `sys/resource.h` as follows:

```
struct rlimit {
long rlim_cur; /* current (soft) limit */
long rlim_max; /* maximum value for rlim_cur */
};
```

The following resource types can be passed in rltype:

`RLIMIT_CPU`
: CPU time in milliseconds.

`RLIMIT_FSIZE`
: Maximum file size.

`RLIMIT_DATA`
: Data size.

`RLIMIT_STACK`
: Stack size.

`RLIMIT_CORE`
: Core file size.

`RLIMIT_RSS`
: Resident set size.

`RLIMIT_MEMLOCK`
: Locked-in-memory address space.

`RLIMIT_NPROC`
: Number of processes.

`RLIMIT_NOFILE`
: Number of open files.

Currently, only the `RLIMIT_STACK` and `RLIMIT_NOFILE` are meaningful: the first returns the value of `_stklen` (See _stklen), the second the value returned by `sysconf(_SC_OPEN_MAX)` (See sysconf). All other members of the `rlimit` structure are set to `RLIM_INFINITY`, defined in `sys/resource.h` as `2^31 - 1`.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct rlimit rlimitbuf;
int rc = getrlimit (RLIMIT_STACK, &rlimitbuf);
```

getrusage

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(int who, struct rusage *rusage);
```

Description

This function returns information about the running process. The structure `struct rusage` is defined on `<sys/resource.h>` as follows:

```
struct rusage {
struct timeval ru_utime; /* user time used */
struct timeval ru_stime; /* system time used */
long ru_maxrss; /* integral max resident set size */
long ru_ixrss; /* integral shared text memory size */
long ru_idrss; /* integral unshared data size */
long ru_isrss; /* integral unshared stack size */
long ru_minflt; /* page reclaims */
long ru_majflt; /* page faults */
long ru_nswap; /* swaps */
long ru_inblock; /* block input operations */
```

```
long ru_oublock; /* block output operations */
long ru_msgsnd; /* messages sent */
long ru_msgrcv; /* messages received */
long ru_nsignals; /* signals received */
long ru_nvcsw; /* voluntary context switches */
long ru_nivcsw; /* involuntary context switches */
};
```

Currently, the only field that is computed is `ru_utime`. It is computed as the total elapsed time used by the calling program. The remainder of the fields are set to zero.

The who parameter must be `RUSAGE_SELF` or `RUSAGE_CHILDREN`.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct rusage r;
getrusage(RUSAGE_SELF, &r);
```

gets

Syntax

```
#include <stdio.h>

char *gets(char *buffer);
```

Description

Reads characters from `stdin`, storing them in buffer, until either end of file or a newline is encountered. If any characters were stored, the buffer is then `NULL` terminated and its address is returned, else `NULL` is returned.

Return Value

The address of the buffer, or `NULL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char buf[1000];
while (gets(buf))
puts(buf);
```

gettext

Syntax

```
#include <conio.h>

int gettext(int _left, int _top, int _right, int _bottom,
void *_destin);
```

Description

Retrieve a block of screen characters into a buffer. `gettext` is a macro defined in `conio.h` that will expand into `_conio_gettext` (See _conio_gettext). This is needed to resolve the name conflict existing between the `gettext` function from `libintl.a` defined in `libintl.h` and this one defined in `conio.h`. If you want to use both `gettext` functions in the same source file you must use `_conio_gettext` (See _conio_gettext) to get the `gettext` function from `conio.h`. This means that if both headers are included in the same source file the `gettext` keyword will always be reserved for the `gettext` function defined in `libintl.h` and indeed will always make reference to the `gettext` function from `libintl.a`.

Return Value

1

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

gettextinfo

Syntax

```
#include <conio.h>

void gettextinfo(struct text_info *_r);
```

Description

This function returns the parameters of the current window on the screen. The return structure is this:

```
struct text_info {
unsigned char winleft;
unsigned char wintop;
unsigned char winright;
unsigned char winbottom;
unsigned char attribute;
unsigned char normattr;
unsigned char currmode;
unsigned char screenheight;
unsigned char screenwidth;
unsigned char curx;
unsigned char cury;
};
```

The `normattr` field is the text attribute which was in effect before the program started.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

gettime

Syntax

```
#include <dos.h>

void gettime(struct time *);
```

Description

This function gets the current time. The return structure is as follows:

```
struct time {
unsigned char ti_min;
unsigned char ti_hour;
unsigned char ti_hund;
unsigned char ti_sec;
};
```

See settime. See getdate.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct time t;
gettime(&t);
```

gettimeofday

Syntax

```
#include <time.h>

int gettimeofday(struct timeval *tp, struct timezone *tzp);
```

Description

Gets the current GMT time and the local timezone information. The return structures are as follows:

```
struct timeval {
long tv_sec; /* seconds since 00:00:00 GMT 1/1/1970 */
long tv_usec; /* microseconds */
};
struct timezone {
int tz_minuteswest; /* west of GMT */
int tz_dsttime; /* set if daylight saving time in affect */
};
```

If either tp or tzp are `NULL`, that information is not provided.

Note that although this function returns microseconds for compatibility reasons, the values are precise to less than 1/20 of a second only. The underlying DOS function has 1/20 second granularity, as it is calculated from the 55 ms timer tick count, so you won't get better than that with gettimeofday().

See settimeofday.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

getuid

Syntax

```
#include <unistd.h>

int getuid(void);
```

Description

Returns the user ID.

Return Value

42

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

getw

Syntax

```
#include <stdio.h>

int getw(FILE *file);
```

Description

Reads a single 32-bit binary word in native format from file. This function is provided for compatibility with other 32-bit environments, so it reads a 32-bit `int`, not a 16-bit `short`, like some 16-bit DOS compilers do.

See putw.

Return Value

The value read, or `EOF` for end-of-file or error. Since `EOF` is a valid integer, you should use `feof` or `ferror` to detect this situation.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int i = getw(stdin);
```

getwd

Syntax

```
#include <unistd.h>

char *getwd(char *buffer);
```

Description

Get the current directory and put it in buffer. The return value includes the drive specifier.

Return Value

buffer is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[PATH_MAX];
getwd(buf);
```

getxkey

Syntax

```
#include <pc.h>
#include <keys.h>
```

```
int getxkey(void);
```

Description

Waits for the user to press one key, then returns that key. Alt-key combinations have 0x100 added to them, and extended keys have 0x200 added to them.

The file `keys.h` has symbolic names for many of the keys.

See getkey.

Return Value

The key pressed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (getxkey() != K_EEnd)
do_something();
```

glob

Syntax

```
#include <glob.h>

int glob(const char *pattern, int flags,
int (*errfunc)(const char *epath, int eerrno), glob_t *pglob);
```

Description

This function expands a filename wildcard which is passed as pattern. The pattern may include these special characters:

`*`
: Matches zero of more characters.

`?`
: Matches exactly one character (any character).

`[...]`
: Matches one character from a group of characters. If the first character is `!`, matches any character *not* in the group. A group is defined as a list of characters between the brackets, e.g. `[dkl_]`, or by two characters separated by `-` to indicate all characters between and including these two. For example, `[a-d]` matches `a`, `b`, `c`, or `d`, and `[!a-zA-Z0-9]` matches any character that is not alphanumeric.

`...`
: Matches all the subdirectories, recursively (VMS aficionados, rejoice!).

`\`
: Causes the next character to not be treated as special. For example, `\[` matches a literal `[`. If flags includes `GLOB_NOESCAPE`, this quoting is disabled and `\` is handled as a simple character.

The variable flags controls certain options of the expansion process. Possible values for flags are as follows:

`GLOB_APPEND`
: Append the matches to those already present in the array `pglob->gl_pathv`. By default, `glob` discards all previous contents of `pglob->gl_pathv` and allocates a new memory block for it. If you use `GLOB_APPEND`, `pglob` should point to a structure returned by a previous call to `glob`.

`GLOB_DOOFFS`
: Skip `pglob->gl_offs` entries in `gl_pathv` and put new matches after that point. By default, `glob` puts the new matches beginning at `pglob->gl_pathv[0]`. You can use this flag both with `GLOB_APPEND` (in which case the new matches will be put after the first `pglob->gl_offs` matches from previous call to `glob`), or without it (in which case the first `pglob->gl_offs` entries in `pglob->gl_pathv` will be

filled by NULL pointers).

GLOB_ERR
: Stop when an unreadable directory is encountered and call user-defined function errfunc. This cannot happen under DOS (and thus errfunc is never used).

GLOB_MARK
: Append a slash to each pathname that is a directory.

GLOB_NOCHECK
: If no matches are found, return the pattern itself as the only match. By default, glob doesn't change pglob if no matches are found.

GLOB_NOESCAPE
: Disable blackslash as an escape character. By default, backslash quotes special meta-characters in wildcards described above.

GLOB_NOSORT
: Do not sort the returned list. By default, the list is sorted alphabetically. This flag causes the files to be returned in the order they were found in the directory.

Given the pattern and the flags, glob expands the pattern and returns a list of files that match the pattern in a structure a pointer to which is passed via pglob. This structure is like this:

```
typedef struct {
size_t gl_pathc;
char **gl_pathv;
size_t gl_offs;
} glob_t;
```

In the structure, the gl_pathc field holds the number of filenames in gl_pathv list; this includes the filenames produced by this call, plus any previous filenames if GLOB_APPEND or GLOB_DOOFFS were set in flags. The list of matches is returned as an array of pointers to the filenames; gl_pathv holds the address of the array. Thus, the filenames which match the pattern can be accessed as gl_pathv[0], gl_pathv[1], etc. If GLOB_DOOFFS was set in flags, the new matches begin at offset given by gl_offs.

glob allocates memory to hold the filenames. This memory should be freed by calling globfree (See globfree).

Return Value

Zero on success, or one of these codes:

GLOB_ABORTED
: Not used in DJGPP implementation.

GLOB_NOMATCH
: No files matched the given pattern.

GLOB_NOSPACE
: Not enough memory to accomodate expanded filenames.

GLOB_ERR
: Never happens on MSDOS, see above.

Notes

glob will not match names of volume labels.

On MSDOS, filenames are always matched case-insensitively. On filesystems that preserve letter-case in filenames (such as Windows 9x), matches are case-insensitive unless the pattern includes uppercase characters.

On MSDOS, the list of expanded filenames will be returned in lower case, if all the characters of the pattern (except those between brackets [...]) are lower-case; if some of them are upper-case, the expanded filenames will be also in upper case. On filesystems that preserve letter-case in filenames, long filenames are returned as they are found in the directory entry; DOS-style 8+3 filenames are returned as on MSDOS (in lower case if the pattern doesn't include any upper-case letters, in upper case otherwise).

When the environment variable LFN is set to **n**, glob behaves on Windows 9x exactly as it does on MSDOS.

Setting the environment variable FNCASE to **y**, or setting the _CRT0_FLAG_PRESERVE_FILENAME_CASE bit in the _crt0_startup_flags variable (See _crt0_startup_flags) suppresses any letter-case conversions in filenames and forces case-sensitive filename matching. See _preserve_fncase.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

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

/* Convert a wildcard pattern into a list of blank-separated
filenames which match the wildcard. */

char * glob_pattern(char *wildcard)
{

char *gfilename;
size_t cnt, length;
glob_t glob_results;
char **p;

glob(wildcard, GLOB_NOCHECK, 0, &glob_results);

/* How much space do we need? */
for (p = glob_results.gl_pathv, cnt = glob_results.gl_pathc;
cnt; p++, cnt--)
length += strlen(*p) + 1;

/* Allocate the space and generate the list. */
gfilename = (char *) calloc(length, sizeof(char));
for (p = glob_results.gl_pathv, cnt = glob_results.gl_pathc;
cnt; p++, cnt--)
{
strcat(gfilename, *p);
if (cnt > 1)
strcat(gfilename, " ");
}

globfree(&glob_results);
return gfilename;
}
```

globfree

Syntax

```
#include <glob.h>

void globfree(glob_t *pglob);
```

Description

Frees the memory associated with pglob, which should have been allocated by a call to `glob` (See glob).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

gmtime

Syntax

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

```
struct tm *gmtime(const time_t *tod);
```

Description

Converts the time represented by tod into a structure.

The return structure has this format:

```
struct tm {
int tm_sec; /* seconds after the minute [0-60] */
int tm_min; /* minutes after the hour [0-59] */
int tm_hour; /* hours since 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 */
long tm_gmtoff; /* offset from GMT in seconds */
char * tm_zone; /* timezone abbreviation */
};
```

Return Value

A pointer to a static structure which is overwritten with each call.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
time_t x;
struct tm *t;
time(&x);
t = gmtime(&x);
```

_go32_conventional_mem_selector

Syntax

```
#include <go32.h>

u_short _go32_conventional_mem_selector();
```

Description

This function returns a selector which has a physical base address corresponding to the beginning of conventional memory. This selector can be used as a parameter to `movedata` (See movedata) to manipulate memory in the conventional address space.

Return Value

The selector.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
short blank_row_buf[ScreenCols()];
/* scroll screen */
movedata(_go32_conventional_mem_selector(), 0xb8000 + ScreenCols()*2,
_go32_conventional_mem_selector(), 0xb8000,
ScreenCols() * (ScreenRows()-1) * 2);
/* fill last row */
movedata(_go32_my_ds, (int)blank_row_buf,
_go32_conventional_mem_selector(),
```

```
0xb8000 + ScreenCols()*(ScreenRows()-1)*2,
ScreenCols() * 2);
```

_go32_dpmi_allocate_dos_memory

Syntax

```
#include <dpmi.h>

int _go32_dpmi_allocate_dos_memory(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

Allocate a part of the conventional memory area (the first 640K). Set the `size` field of info to the number of paragraphs requested (this is (size in bytes + 15)/16), then call. The `rm_segment` field of info contains the segment of the allocated memory.

The memory may be resized with `_go32_dpmi_resize_dos_memory` and must be freed with `_go32_dpmi_free_dos_memory`.

If there isn't enough memory in the system, the `size` field of info has the largest available size, and an error is returned.

See also See dosmemput, and See dosmemget.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_seginfo info;
info.size = (want_size+15) / 16;
_go32_dpmi_allocate_dos_memory(&info);
dosmemput(buffer, want_size, info.rm_segment*16);
_go32_dpmi_free_dos_memory(&info);
```

_go32_dpmi_allocate_iret_wrapper

Syntax

```
#include <dpmi.h>

int _go32_dpmi_allocate_iret_wrapper(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

This function creates a small assembler function that handles the overhead of servicing an interrupt. To use, put the address of your servicing function in the `pm_offset` field of info and call this function. The `pm_field` will get replaced with the address of the wrapper function, which you pass to both `_go32_dpmi_set_protected_mode_interrupt_vector` and `_go32_dpmi_free_iret_wrapper`.

Warning! Because of the way DPMI works, you may *not* `longjmp` out of an interrupt handler or perform any system calls (such as `printf`) from within an interrupt handler.

Do not enable interrupts with `enable()` or `asm("sti")` in your function.

See also See _go32_dpmi_set_protected_mode_interrupt_vector, and See _go32_dpmi_free_iret_wrapper.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_seginfo info;
info.pm_offset = my_handler;
_go32_dpmi_allocate_iret_wrapper(&info);
_go32_dpmi_set_protected_mode_interrupt_handler(0x75, &info);
...
_go32_dpmi_free_iret_wrapper(&info);
```

_go32_dpmi_allocate_real_mode_callback_iret

Syntax

```
#include <dpmi.h>

int _go32_dpmi_allocate_real_mode_callback_iret(
_go32_dpmi_seginfo *info, _go32_dpmi_registers *regs
);
```

Description

See DPMI Overview.

This function allocates a "real-mode callback". Fill in the `pm_offset` field of info and call this function. It will fill in the `rm_segment` and `rm_offset` fields. Any time a real-mode program calls the real-mode address, your function gets called. The registers in effect will be stored in regs, which should be a global, and will be passed to your function. Any changes in regs will be reflected back into real mode. A wrapper will be added to your function to simulate the effects of an `iret` instruction, so this function is useful for trapping real-mode software interrupts (like 0x1b - **Ctrl-Break** hit).

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_registers regs;

my_handler(_go32_dpmi_registers *r)
{

r->d.eax = 4;
}


setup()
{

_go32_dpmi_seginfo info;
_go32_dpmi_seginfo old_vector;
_go32_dpmi_get_real_mode_interrupt_vector(0x84, &old_vector);
info.pm_offset = my_handler;
_go32_dpmi_allocate_real_mode_callback_iret(&info, &regs);
_go32_dpmi_set_real_mode_interrupt_vector(0x84, &info);
do_stuff();
_go32_dpmi_set_real_mode_interrupt_vector(0x84, &old_vector);
_go32_dpmi_free_real_mode_callback(&info);
}
```

_go32_dpmi_allocate_real_mode_callback_retf

Syntax

```
#include <dpmi.h>

int _go32_dpmi_allocate_real_mode_callback_retf(
_go32_dpmi_seginfo *info, _go32_dpmi_registers *regs
);
```

Description

See DPMI Overview.

This function allocates a "real-mode callback". Fill in the `pm_offset` field of info and call this function. It will fill in the `rm_segment` and `rm_offset` fields. Any time a real-mode program calls the real-mode address, your function gets called. The registers in effect will be stored in regs, which should be a global, and will be passed to your function. Any changes in regs will be reflected back into real mode. A wrapper will be added to your function to simulate the effects of a far return, such as the callback for the packet driver receiver.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See _go32_dpmi_allocate_real_mode_callback_iret, for an example of usage.

_go32_dpmi_chain_protected_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int _go32_dpmi_chain_protected_mode_interrupt_vector(
int vector, _go32_dpmi_seginfo *info
);
```

Description

See DPMI Overview.

This function is used to chain a protected mode interrupt. It will build a suitable wrapper that will call your function and then jump to the next handler. Your function need not perform any special handling.

Warning! Because of the way DPMI works, you may *not* `longjmp` out of an interrupt handler or perform any system calls (such as `printf`) from within an interrupt handler.

Do not enable interrupts with `enable()` or `asm("sti")` in your function.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See _go32_dpmi_set_protected_mode_interrupt_vector.

_go32_dpmi_free_dos_memory

Syntax

```
#include <dpmi.h>

int _go32_dpmi_free_dos_memory(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

This function frees the conventional memory allocated by `_go32_dpmi_allocate_real_mode_memory`. You should pass it the same structure as was used to allocate it.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_seginfo info;
info.size = 100;
_go32_dpmi_allocate_dos_memory(&info);
_go32_dpmi_free_dos_memory(&info);
```

_go32_dpmi_free_iret_wrapper

Syntax

```
#include <dpmi.h>

int _go32_dpmi_free_iret_wrapper(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

This function frees the memory used by the wrapper created by `_go32_dpmi_allocate_iret_wrapper`. You should not free a wrapper that is still in use.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See _go32_dpmi_allocate_iret_wrapper.

_go32_dpmi_free_real_mode_callback

Syntax

```
#include <dpmi.h>

int _go32_dpmi_free_real_mode_callback(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

This function frees the real-mode callbacks and wrappers allocated by `_go32_dpmi_allocate_real_mode_callback_iret` and `_go32_dpmi_allocate_real_mode_callback_retf`.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See _go32_dpmi_allocate_real_mode_callback_iret, for an example of usage.

_go32_dpmi_get_free_memory_information

Syntax

```
#include <dpmi.h

int _go32_dpmi_get_free_memory_information(_go32_dpmi_meminfo *info);
```

Description

This function fills in the following structure:

```
typedef struct {
u_long available_memory;
u_long available_pages;
u_long available_lockable_pages;
u_long linear_space;
u_long unlocked_pages;
u_long available_physical_pages;
u_long total_physical_pages;
u_long free_linear_space;
u_long max_pages_in_paging_file;
u_long reserved[3];
} _go32_dpmi_meminfo;
```

The only field that is guaranteed to have useful data is `available_memory`. Any unavailable field has -1 in it.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int phys_mem_left()
{

_go32_dpmi_meminfo info;
_go32_dpmi_get_free_memory_information(&info);
if (info.available_physical_pages != -1)
return info.available_physical_pages * 4096;
return info.available_memory;
}
```

_go32_dpmi_get_protected_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int _go32_dpmi_get_protected_mode_interrupt_vector(
int vector, _go32_dpmi_seginfo *info
);
```

Description

See DPMI Overview.

This function puts the selector and offset of the specified interrupt vector into the `pm_selector` and `pm_offset` fields of info. This structure can be saved and later passed to `_go32_dpmi_set_protected_mode_interrupt_vector` to restore a vector.

Return Value
Zero on success, nonzero on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
See _go32_dpmi_set_protected_mode_interrupt_vector, for an example of usage.

_go32_dpmi_get_real_mode_interrupt_vector
Syntax
```
#include <dpmi.h>

int _go32_dpmi_get_real_mode_interrupt_vector(
int vector, _go32_dpmi_seginfo *info
);
```

Description
See DPMI Overview.

This function gets the real-mode interrupt vector specified into the address in the `rm_segment` and `rm_offset` fields in info.

Return Value
Zero on success, nonzero on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
See _go32_dpmi_allocate_real_mode_callback_iret, for an example of usage.

_go32_dpmi_lock_code
Syntax
```
#include <dpmi.h>

int _go32_dpmi_lock_code( void *lockaddr, unsigned long locksize);
```

Description
Locks the given region of code, starting at lockaddr for locksize bytes. lockaddr is a regular pointer in your program, such as the address of a function.

Return Value
0 if success, -1 if failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
void my_handler()
{

}
```

```
void lock_my_handler()
{

_go32_dpmi_lock_code(my_handler,
(unsigned long)(lock_my_handler - my_handler));
}
```

_go32_dpmi_lock_data

Syntax

```
#include <dpmi.h>

int _go32_dpmi_lock_data( void *lockaddr, unsigned long locksize);
```

Description

Locks the given region of data, starting at lockaddr for locksize bytes. lockaddr is a regular pointer in your program, such as the address of a variable.

Return Value

0 if success, -1 if failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int semaphore=0;

void lock_my_handler()
{

_go32_dpmi_lock_data(&semaphore, 4);
}
```

_go32_dpmi_remaining_physical_memory

Syntax

```
#include <dpmi.h>

unsigned long _go32_dpmi_remaining_physical_memory(void);
```

Description

Returns the amount of physical memory that is still available in the system.

Return Value

The amount in bytes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_dpmi_remaining_virtual_memory

Syntax

```
#include <dpmi.h>

unsigned long _go32_dpmi_remaining_virtual_memory(void);
```

Description

Returns the amount of virtual memory that is still available in the system.

Return Value

The amount in bytes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_dpmi_resize_dos_memory

Syntax

```
#include <dpmi.h>

int _go32_dpmi_resize_dos_memory(_go32_dpmi_seginfo *info);
```

Description

See DPMI Overview.

The info structure is the same one used to allocate the memory. Fill in a new value for `size` and call this function. If there is not enough memory to satisfy the request, the largest size is filled in to the `size` field, the memory is not resized, and this function fails.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_seginfo info;
info.size = 10;
_go32_dpmi_allocate_dos_memory(&info);
info.size = 20;
_go32_dpmi_resize_dos_memory(&info);
_go32_dpmi_free_dos_memory(&info);
```

_go32_dpmi_set_protected_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int _go32_dpmi_set_protected_mode_interrupt_vector(
int vector, _go32_dpmi_seginfo *info
);
```

Description

See DPMI Overview.

This function sets the protected mode interrupt vector specified to point to the given function. The `pm_offset` and `pm_selector` fields of info must be filled in (See _go32_my_cs). The following should be noted:

- You may not `longjmp` out of an interrupt handler.
- You may not make any function calls that require system calls, such as `printf`.
- This function will not wrap the handler for you. The `_go32_dpmi_allocate_iret_wrapper` and `_go32_dpmi_chain_protected_mode_interrupt_vector` functions can wrap your function if you want.
- You must set the pm_selector field of info. Use `_go32_my_cs` to get a selector valid for your functions.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
volatile int tics = 0;

timer_handler()
{

tics++;
}


int main()
{

_go32_dpmi_seginfo old_handler, new_handler;

printf("grabbing timer interrupt\n");
_go32_dpmi_get_protected_mode_interrupt_vector(8, &old_handler);
new_handler.pm_offset = (int)tic_handler;
new_handler.pm_selector = _go32_my_cs();
_go32_dpmi_chain_protected_mode_interrupt_vector(8, &new_handler);

getkey();

printf("releasing timer interrupt\n");
_go32_dpmi_set_protected_mode_interrupt_vector(8, &old_handler);

return 0;
}
```

_go32_dpmi_set_real_mode_interrupt_vector

Syntax

```
#include <dpmi.h>

int _go32_dpmi_set_real_mode_interrupt_vector(
int vector, _go32_dpmi_seginfo *info
);
```

Description

See DPMI Overview.

This function sets the real-mode interrupt vector specified to point to the address in the `rm_segment` and `rm_offset` fields in info.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See _go32_dpmi_allocate_real_mode_callback_iret, for an example of usage.

_go32_dpmi_simulate_fcall

Syntax

```
#include <dpmi.h>

int _go32_dpmi_simulate_fcall(_go32_dpmi_registers *regs);
```

Description

See DPMI Overview.

This function simulates a real-mode far call to a function that returns with a far return. The registers are set up from regs, including `CS` and `IP`, which indicate the address of the call. Any registers the function modifies are reflected in regs on return.

If `SS` and `SP` are both zero, a small temporary stack is used when in real mode. If not, they are used *as is*. It's a good idea to use `memset` to initialize the register structure before using it.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_registers r;
r.x.ax = 47;
r.x.cs = some_segment;
r.x.ip = some_offset;
r.x.ss = r.x.sp = 0;
_go32_dpmi_simulate_fcall(&r);
printf("returns %d\n", r.x.ax);
```

_go32_dpmi_simulate_fcall_iret

Syntax

```
#include <dpmi.h>

int _go32_dpmi_simulate_fcall_iret(_go32_dpmi_registers *regs);
```

Description

See DPMI Overview.

This function simulates a real-mode far call to a function that returns with an `iret` instruction. The registers are set up from regs, including `CS` and `IP`, which indicate the address of the call. Any registers the function modifies are reflected in regs on return.

If `SS` and `SP` are both zero, a small temporary stack is used when in real mode. If not, they are used *as is*. It's a good idea to use `memset` to initialize the register structure before using it.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_registers r;
r.x.ax = 47;
r.x.cs = some_segment;
r.x.ip = some_offset;
r.x.ss = r.x.sp = 0;
_go32_dpmi_simulate_fcall_iret(&r);
printf("returns %d\n", r.x.ax);
```

_go32_dpmi_simulate_int

Syntax

```
#include <dpmi.h>

int _go32_dpmi_simulate_int(int vector, _go32_dpmi_registers *regs);
```

Description

See DPMI Overview.

This function simulates a real-mode interrup. The registers are set up from regs, including `CS` and `IP`, which indicate the address of the call. Any registers the function modifies are reflected in regs on return.

If `SS` and `SP` are both zero, a small temporary stack is used when in real mode. If not, they are used *as is*. It's a good idea to use `memset` to initialize the register structure before using it.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_go32_dpmi_registers r;
r.h.ah = 0x08;
r.h.dl = 0x80; /* drive C: */
r.x.ss = r.x.sp = 0;
_go32_dpmi_simulate_int(0x13, &r);
printf("disk is %d cyl, %d head, %d sect\n",
r.h.ch | ((r.x.cl<<2)&0x300),
r.h.dh, r.h.cl & 0x3f));
```

_go32_info_block

Syntax

```
#include <go32.h>

extern __Go32_Info_Block _go32_info_block;
```

Description

The go32 information block is a mechanism for `go32` to pass information to the application. Some of this information is generally useful, such as the pid or the transfer buffer, while some is used internally to `libc.a` only.

The structure has this format:

```
typedef struct {
unsigned long size_of_this_structure_in_bytes;
unsigned long linear_address_of_primary_screen;
unsigned long linear_address_of_secondary_screen;
unsigned long linear_address_of_transfer_buffer;
unsigned long size_of_transfer_buffer;
unsigned long pid;
unsigned char master_interrupt_controller_base;
unsigned char slave_interrupt_controller_base;
unsigned short selector_for_linear_memory;
unsigned long linear_address_of_stub_info_structure;
unsigned long linear_address_of_original_psp;
unsigned short run_mode;
unsigned short run_mode_info;
} Go32_Info_Block;
```

The linear address fields provide values that are suitable for `dosmemget`, `dosmemput`, and `movedata`. The selector_for_linear_memory is suitable for `<sys/farptr.h>` selector parameters.

Due to the length of these fields, and their popularity, the following macros are available:

`_dos_ds`
: This expands to _go32_info_block.selector_for_linear_memory

`__tb`
: This expands to _go32_info_block.linear_address_of_transfer_buffer

`__tb_size`
: This expands to _go32_info_block.size_of_transfer_buffer

The `run_mode` field indicates the mode that the program is running in. The following modes are defined:

`_GO32_RUN_MODE_UNDEF`
: This indicates that the extender did not (or could not) determine or provide the mode information. The most probable reason is that it's an older extender that does not support this field. The program should not assume anything about the run mode if it is this value.

`_GO32_RUN_MODE_RAW`
: This indicates that no CPU manager is being used, and no XMS manager is present. The CPU is being managed directly from the extender, and memory was allocated from the extended memory pool.

`_GO32_RUN_MODE_XMS`
: This indicates that the extender is managing the CPU, but an XMS driver is managing the memory pool.

`_GO32_RUN_MODE_VCPI`
: This indicates that a VCPI server (like `emm386` or `qemm`) is managing both the CPU and the memory.

`_GO32_RUN_MODE_DPMI`
: This indicates that a DPMI server (like `qdpmi` or Windows) is managing both the CPU and memory. Programs may rely on this value to determine if it is safe to use DPMI 0.9 functions.

  If this value is set, the `run_mode_info` field has the DPMI specification version, in hex, shifted eight bits. For example, DPMI 0.9 has 0x005A in the `run_mode_info` field.

Note that the program should not assume that the value will be one of the listed values. If the program is running with an extender that provides some other mode (say, a newly released extender) then the program should be able to handle that case gracefully.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
dosmemget(_go32_info_block.linear_address_of_primary_screen,
80*25*2, buf);
```

_go32_interrupt_stack_size

Syntax

```
#include <dpmi.h>

extern unsigned long _go32_interrupt_stack_size;
```

Description

The default size of the interrupt handler's stack. Defaults to 32k.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_my_cs

Syntax

```
#include <go32.h>

u_short _go32_my_cs();
```

Description

Returns the current `CS`. This is useful for setting up interrupt vectors and such.

Return Value

`CS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_my_ds

Syntax

```
#include <go32.h>

u_short _go32_my_ds();
```

Description

Returns the current `DS`. This is useful for moving memory and such.

Return Value

`DS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_my_ss

Syntax

```
#include <go32.h>

u_short _go32_my_ss();
```

Description

Returns the current `SS`. This is useful for moving memory and such.

Return Value

`SS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_rmcb_stack_size

Syntax

```
#include <dpmi.h>

extern unsigned long _go32_rmcb_stack_size;
```

Description

The default size of the real mode callback handler's stack. Defaults to 32k.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_go32_want_ctrl_break

Syntax

```
#include <go32.h>

void _go32_want_ctrl_break(int yes);
```

Description

This function tells go32 whether or not it wants **Ctrl-Break** to be an exception or passed to the application. If you pass a nonzero value for yes, pressing **Ctrl-Break** will set a flag that can be detected with `_go32_was_ctrl_break_hit` (See _go32_was_ctrl_break_hit). If you pass zero for yes, when you press **Ctrl-Break** the program will be terminated.

Note that if you call `_go32_was_ctrl_break_hit`, this function automatically gets called to ask for **Ctrl-Break** events.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
_g32_want_ctrl_break(1);
do_something_long();
_g32_want_ctrl_break(0);
```

_go32_was_ctrl_break_hit

Syntax

```
#include <go32.h>

unsigned _go32_was_ctrl_break_hit(void);
```

Description

This function returns the number of times that **Ctrl-Break** was hit since the last call to this function or `_go32_want_ctrl_break` (See _go32_want_ctrl_break).

Return Value

Zero if **Ctrl-Break** hasn't been hit, nonzero to indicate how many times if it has been hit.

Note that `_go32_want_ctrl_break` is automatically called to request these events, so you don't have to set up for this call.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (!_go32_was_ctrl_break_hit())
do_something();
```

gotoxy

Syntax

```
#include <conio.h>

void gotoxy(int x, int y);
```

Description

Move the cursor to row y, column x. The upper left corner of the current window is (1,1).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

gppconio_init

Syntax

```
#include <conio.h>

void gppconio_init(void);
```

Description

Initialize the screen. This is called automatically at program start-up if you use any of the `conio` functions, but there may be times when you need to call it again, typically after calling some video BIOS function which affects screen parameters.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__has_fd_properties

Syntax

```
#include <libc/fd_props.h>

int __has_fd_properties(int fd);
```

Description

This internal function checks whether there are any properties associated with the file descriptor fd.

Return Value

Non-zero if fd has any properties; zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

hasmntopt

Syntax

```
#include <mntent.h>

char *hasmntopt(const struct mntent *mnt, const char *opt);
```

Description

This function scans the `mnt_opts` field of the `mntent` structure pointed to by mnt for a substring that matches opt. See getmntent.

Return Value

This function returns the address of the substring if a match is found, or `NULL` otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

highvideo

Syntax

```
#include <conio.h>

void highvideo(void);
```

Description

Causes any new characters put on the screen to be bright.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

htonl

Syntax

```
#include <netinet/in.h>

unsigned long htonl(unsigned long val);
```

Description

This function converts from host formatted longs to network formatted longs. For the i386 and higher processors, this means that the bytes are swapped from 1234 order to 4321 order.

Return Value

The network-order value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
packet.ipaddr = htonl(ip);
```

htons

Syntax

```
#include <netinet/in.h>

unsigned short htons(unsigned short val);
```

Description

This function converts from host formatted shorts to network formatted shorts. For the i386 and higher processors, this means that the bytes are swapped from 12 order to 21 order.

Return Value

The network-order value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
tcp.port = htons(port);
```

hypot

Syntax

```
#include <math.h>

double hypot(double x, double y);
```

Description

This function computes `sqrt(x*x + y*y)`, the length of a hypotenuse of a right triangle whose shorter sides are x and y. In other words, it computes the Euclidean distance between the points `(0,0)` and `(x,y)`. Since the computation is done in extended precision, there is no danger of overflow or underflow when squaring the arguments, whereas direct computation of `sqrt(x*x + y*y)` could cause overflow or underflow for extreme (very large or very small) values of x and y.

Return Value

The value of `sqrt(x*x + y*y)`. If both arguments are finite, but the result is so large that it would overflow a `double`, the return value is `Inf`, and `errno` is set to `ERANGE`. If one of the arguments is `Inf`, the return value is `Inf` and the value of `errno` is left unchanged. If one of the arguments is `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

imaxabs

Syntax

```
#include <inttypes.h>

intmax_t imaxabs (intmax_t x);
```

Description

This function takes the absolute value of x.

`abs` (See abs) operates on an `int`. This function operates on a greatest-width integer.

Return Value

`|x|`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

imaxdiv

Syntax

```
#include <inttypes.h>

imaxdiv_t imaxdiv (intmax_t numerator, intmax_t denominator);
```

Description

Returns the quotient and remainder of the division numerator divided by denominator. The return type is as follows:

```
typedef struct {
intmax_t quot;
intmax_t rem;
} imaxdiv_t;
```

Return Value

The results of the division are returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
imaxdiv_t l = imaxdiv(42, 3);
printf("42 = %" PRIdMAX " x 3 + %" PRIdMAX "\n", l.quot, l.rem);

imaxdiv(+40, +3) = { +13, +1 }
imaxdiv(+40, -3) = { -13, -1 }
imaxdiv(-40, +3) = { -13, -1 }
imaxdiv(-40, -3) = { +13, -1 }
```

inb

Syntax

```
#include <pc.h>

unsigned char inb(unsigned short _port);
```

Description

Calls See inportb. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

index

Syntax

```
#include <strings.h>

char *index(const char *string, int ch);
```

Description

Returns a pointer to the first occurrence of ch in string. Note that the `NULL` character counts, so if you pass zero as ch you'll get a pointer to the end of the string back.

Return Value

A pointer to the character, or `NULL` if it wasn't found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (index(path, '*'))
do_wildcards(path);
```

initstate

Syntax

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

```
char *initstate(unsigned seed, char *arg_state, int n);
```

Description

Initializes the random number generator (See random) with pointer arg_state to array of n bytes, then calls `srandom` with seed.

Return Value

Pointer to old state information.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inp

Syntax

```
#include <pc.h>

unsigned char inp(unsigned short _port);
```

Description

Calls See inportb. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportb

Syntax

```
#include <pc.h>

unsigned char inportb(unsigned short _port);
```

Description

Read a single 8-bit I/O port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Return Value

The value returned through the port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportl

Syntax

```
#include <pc.h>

unsigned long inportl(unsigned short _port);
```

Description

This function reads a single 32-bit I/O port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Return Value

The value returned from the port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportsb

Syntax

```
#include <pc.h>

void inportsb(unsigned short _port, unsigned char *_buf, unsigned _len);
```

Description

Reads the 8-bit _port _len times, and stores the bytes in buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportsl

Syntax

```
#include <pc.h>

void inportsl(unsigned short _port, unsigned long *_buf, unsigned _len);
```

Description

Reads the 32-bit _port _len times, and stores the bytes in buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportsw

Syntax

```
#include <pc.h>

void inportsw(unsigned short _port,
unsigned short *_buf, unsigned _len);
```

Description

Reads the 16-bit _port _len times, and stores the bytes in buf.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inportw

Syntax

```
#include <pc.h>

unsigned short inportw(unsigned short _port);
```

Description

Read a single 16-bit I/O port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Return Value

The value returned through the port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

inpw

Syntax

```
#include <pc.h>

unsigned short inpw(unsigned short _port);
```

Description

Calls See inportw. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

insline

Syntax

```
#include <conio.h>

void insline(void);
```

Description

A blank line is inserted at the current cursor position. The previous line and lines below it scroll down.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

insque

Syntax

```
#include <search.h>

void insque(struct qelem *elem, struct qelem *pred);
```

Description

This function manipulates queues built from doubly linked lists. Each element in the queue must be in the form of `struct qelem` which is defined thus:

```
struct qelem {
struct qelem *q_forw;
struct qelem *q_back;
char q_data[0];
}
```

This function inserts elem in a queue immediately after pred.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__install_dev_full

Syntax

```
#include <sys/xdevices.h>

int __install_dev_full (void);
```

Description

This function activates support for the special file `/dev/full`. When read, `dev/full` always returns `\0` characters. Writes to `/dev/full` will fail with `errno` set to `ENOSPC`. Seeks on `/dev/full` always succeed.

The DJGPP debug support functions will interfere with `/dev/full` (See File System Extensions).

Return Value

On success, a non-zero value is returned; on failure, zero is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__install_dev_zero

Syntax

```
#include <sys/xdevices.h>

int __install_dev_zero (void);
```

Description

This function activates support for the special file `/dev/zero`. When read, `dev/zero` always returns `\0` characters. When written, `/dev/zero` discards the data. Seeks on `/dev/zero` will always succeed.

The DJGPP debug support functions will interfere with `/dev/zero` (See File System Extensions).

Return Value

On success, a non-zero value is returned; on failure, zero is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

int386

Syntax

```
#include <dos.h>

int int386(int ivec, union REGS *in, union REGS *out);
```

Description

This function is equal to `int86` function. See See int86, for further description.

Return Value

The returned value of `EAX`.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

int386x
Syntax
```
#include <dos.h>

int int386x(int ivec, union REGS *in, union REGS *out,
struct SREGS *seg);
```

Description
This function is equal to `int86x`. See See int86, for further description.

Return Value
The value of `EAX` is returned.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

int86
Syntax
```
#include <dos.h>

int int86(int ivec, union REGS *in, union REGS *out);
```

Description
The `union REGS` is defined by `<dos.h>` as follows:

```
struct DWORDREGS {
unsigned long edi;
unsigned long esi;
unsigned long ebp;
unsigned long cflag;
unsigned long ebx;
unsigned long edx;
unsigned long ecx;
unsigned long eax;
unsigned short eflags;
};

struct DWORDREGS_W {
unsigned long di;
unsigned long si;
unsigned long bp;
unsigned long cflag;
unsigned long bx;
unsigned long dx;
unsigned long cx;
unsigned long ax;
unsigned short flags;
};

struct WORDREGS {
unsigned short di, _upper_di;
unsigned short si, _upper_si;
unsigned short bp, _upper_bp;
unsigned short cflag, _upper_cflag;
unsigned short bx, _upper_bx;
unsigned short dx, _upper_dx;
unsigned short cx, _upper_cx;
```

```
unsigned short ax, _upper_ax;
unsigned short flags;
};

struct BYTEREGS {
unsigned short di, _upper_di;
unsigned short si, _upper_si;
unsigned short bp, _upper_bp;
unsigned long cflag;
unsigned char bl;
unsigned char bh;
unsigned short _upper_bx;
unsigned char dl;
unsigned char dh;
unsigned short _upper_dx;
unsigned char cl;
unsigned char ch;
unsigned short _upper_cx;
unsigned char al;
unsigned char ah;
unsigned short _upper_ax;
unsigned short flags;
};

union REGS {
struct DWORDREGS d;
#ifdef _NAIVE_DOS_REGS
struct WORDREGS x;
#else
#ifdef _BORLAND_DOS_REGS
struct DWORDREGS x;
#else
struct DWORDREGS_W x;
#endif
#endif
struct WORDREGS w;
struct BYTEREGS h;
};
```

Note: The `.x.` branch is a problem generator. Most programs expect the `.x.` branch to have e.g. "`.x.ax`" members, and that they are 16-bit. If you know you want 32-bit values, use the `.d.eax` members. If you know you want 16-bit values, use the `.w.ax` members. The `.x.` members behave according to `#defines`, as follows:

`default`
: If you specify no `#define`, the `.x.` branch has "`ax`" members and is 32-bit. This is compatible with previous versions of djgpp.

`_NAIVE_DOS_REGS`
: This define gives you `.x.ax`, but they are 16-bit. This is probably what most programs ported from 16-bit dos compilers will want.

`_BORLAND_DOS_REGS`
: This define gives you `.x.eax` which are 32-bit. This is compatible with Borland's 32-bit compilers.

This function simulates a software interrupt. Note that, unlike the `__dpmi_int` function, requests that go through `int86` and similar functions are specially processed to make them suitable for invoking real-mode interrupts from protected-mode programs. For example, if a particular routine takes a pointer in `BX`, `int86` expects you to put a (protected-mode) pointer in `EBX`. Therefore, `int86` should have specific support for every interrupt and function you invoke this way. Currently, it supports only a subset of all available interrupts and functions:

1) All functions of any interrupt which expects only scalar arguments registers (i.e., no pointers to buffers).

2) In addition, the following functions of interrupt 21h are supported: 9, 39h, 3Ah, 3Bh, 3Ch, 3Dh, 3Fh, 40h, 41h, 43h, 47h, 56h.

When the interrupt is invoked, the CPU registers are copied from *in*. After the interrupt, the CPU registers are copied to *out*.

This function is just like `int86x` (See int86x) except that suitable default values are used for the segment registers.

See also See int86x, See intdos, and See bdos.

Return Value

The returned value of `EAX`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
union REGS r;
r.x.ax = 0x0100;
r.h.dl = 'c';
int86(0x21, &r, &r);
```

int86x

Syntax

```
#include <dos.h>

int int86x(int ivec, union REGS *in, union REGS *out,
struct SREGS *seg);
```

Description

The `struct SREGS` is defined by `<dos.h>` as follows:

```
struct SREGS {
unsigned short es;
unsigned short ds;
unsigned short fs;
unsigned short gs;
unsigned short cs;
unsigned short ss;
};
```

This function is just like `int86` (See int86) except that values you pass in seg are used for the segment registers instead of the defaults.

See also See int86, See intdos, and See bdos.

Return Value

The value of `EAX` is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
union REGS r;
struct SREGS s;
r.h.ah = 0x31;
r.h.dl = 'c';
r.x.si = si_val;
s.ds = ds_val;
int86x(0x21, &r, &r, &s);
```

intdos

Syntax

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

```
int intdos(union REGS *in, union REGS *out);
```

Description

This function is just like `int86` (See int86x) except that the interrupt vector is 0x21.

Return Value

`EAX`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

intdosx

Syntax

```
#include <dos.h>

int intdosx(union REGS *in, union REGS *out, struct SREGS *s);
```

Description

This function is just like `int86x` (See int86x) except that the interrupt vector is 0x21.

Return Value

`EAX`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

intensevideo

Syntax

```
#include <conio.h>

void intensevideo(void);
```

Description

Bit 7 (`MSB`) of the character attribute byte has two possible effects on EGA and VGA displays: it can either make the character blink or change the background color to bright (thus allowing for 16 background colors as opposed to the usual 8). This function sets that bit to display bright background colors. After a call to this function, every character written to the screen with bit 7 of the attribute byte set, will have a bright background color. The companion function `blinkvideo` (See blinkvideo) has the opposite effect.

Note that there is no BIOS function to get the current status of this bit, but bit 5 of the byte at `0040h:0065h` in the BIOS area indicates the current state: if it's 1 (the default), blinking characters will be displayed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__internal_readlink

Syntax

```
#include <libc/symlink.h>

int __internal_readlink(const char * path, int fhandle,
char * buf, size_t max)
```

Description

In general applications shouldn't call this function; use `readlink` instead (See readlink). However, there is one exception: if you have a `FSEXT` `fstat` file handler, and do not want do anything special about symlinks. In this case you should call this function from your handler to set properly `S_IFLNK` bit in `st_mode`. This function operates on either path **or** fhandle. In any case, the other arg should be set to `NULL` or 0.

Return Value

Number of copied characters; value -1 is returned in case of error and `errno` is set. When value returned is equal to size, you cannot determine if there was enough room to copy whole name. So increase size and try again.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[FILENAME_MAX + 1];
if (__internal_readlink(0, "/dev/env/DJDIR/bin/sh.exe",
buf, FILENAME_MAX) == -1)
if (errno == EINVAL)
puts("/dev/env/DJDIR/bin/sh.exe is not a symbolic link.");
```

_invent_inode

Syntax

```
ino_t
_invent_inode(const char *name, unsigned time_stamp,
unsigned long fsize)
```

Description

This invents an inode number for those files which don't have valid DOS cluster number. These could be:

- devices like `/dev/null` or file system extensions (See File System Extensions)
- empty files which were not allocated disk space yet
- or files on networked drives, for which the redirector doesn't bring the cluster number.

To ensure proper operation of this function, you must call it with a filename in some canonical form. E.g., with a name returned by `truename()` (See _truename), or that returned by `_fixpath()` (See _fixpath). The point here is that the entire program *must* abide by these conventions through its operation, or else you risk getting different inode numbers for the same file.

Return Value

0 on error, otherwise the invented inode number for the file.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ioctl (DOS)

The DOSish version of `ioctl` performs an interrupt 0x21, function 0x44. It takes care of supplying transfer buffers in low address regions, if they are needed. For an exhaustive description of the various commands and subcommands, see Ralf Brown's interrupt list.

It is highly recommended to use only the DOS_* functions listed in `sys/ioctl.h`.

Syntax

```
#include <sys/ioctl.h>

int ioctl(int fd, int cmd, ... );
```

Description

The parameter `fd` must refer to a file descriptor for character device functions, or the number of a block device (usually current=0, A:=1, ...).

The following constants can be used for the `cmd` parameter:

DOS_GETDEVDATA
: Get device information. Returns the device information word from DX. The call to ioctl should look like this:

```
int ret_val = ioctl (fd, DOS_GETDEVDATA);
```

For another way of achieving the same effect, see _get_dev_info (See _get_dev_info).

DOS_SETDEVDATA
: Set device information. Returns the new device information word form DX or -1. The call to ioctl should look like this:

```
int ret_val = ioctl (fd, DOS_SETDEVDATA, 0, dev_info);
```

DOS_RCVDATA
: Read from character device control channel. After cmd must follow the number of requested bytes to read and a pointer to a buffer. Returns the number of bytes actually read or -1 on error. The call to ioctl should look like this:

```
unsigned char buf[bytes_to_read];
int ret_val = ioctl (fd, DOS_RCVDATA, bytes_to_read, &buf);
```

DOS_SNDDATA
: Write to character device control channel. After cmd must follow the number of bytes to write and a pointer to a buffer holding the data. Returns the number of bytes actually written. An example of a call:

```
unsigned char buf[bytes_to_write];
int ret_val = ioctl (fd, DOS_SNDDATA, bytes_to_write, &buf);
```

DOS_RCVCTLDATA
: Read from block device control channel. See DOS_RCVDATA.

DOS_SNDCTLDATA
: Write to block device control channel. See DOS_SNDDATA.

DOS_CHKINSTAT
: Check the input status of a file. Returns 0 if not ready of at EOF, 0xff if file is ready. Here's an example of how to call:

```
int ret_val = ioctl (fd, DOS_CHKINSTAT);
```

A more portable way of doing this is by calling select. See select.

DOS_CHKOUTSTAT
: Check the output status of a file. Returns 0 if not ready of at EOF, 0xff if file is ready. select (See select) is another, more portable way of doing the same.

DOS_ISCHANGEABLE
: Check if a block device is changeable. Returns 0 for removable or 1 for fixed. An example of a call:

```
int ret_val = ioctl (fd, DOS_ISCHANGEABLE);
```

DOS_ISREDIRBLK
: Check if a block device is remote o local. The function _is_remote_drive (See _is_remote_drive) is another way of returning the same info.

DOS_ISREDIRHND
: Check if a file handle refers to a local or remote device. See _is_remote_handle (See _is_remote_handle) for another way of doing this.

DOS_SETRETRY
: Set the sharing retry count. The first extra parameter specifies the pause between retries, the second number of retries. An example:

```
int ret_val = ioctl (fd, DOS_SETRETRY, pause_between_retries,
max_retries);
```

DOS_GENCHARREQ

Generic character device request. Example:

```
int ret_val = ioctl (fd, DOS_GENCHARREQ, category_and_function,
&param_block, si_value, di_value,
param_block_size);
```

Refer to Ralf Brown's Interrupt List for the details about each function and relevant parameter block layout.

`DOS_GENBLKREQ`
Generic block device request. Example of the call:

```
int ret_val = ioctl (drive_no, DOS_GENBLKREQ, category_and_function,
&param_block, si_value, di_value,
param_block_size);
```

Note that instead of the handle, the first argument is the disk drive number (0 = default, 1 = A:, etc.).

`DOS_GLDRVMAP`
Get logical drive map. A call like the following:

```
int ret_val = ioctl (drive_no, DOS_GLDRVMAP);
```

will return 0 if the block device has only one logical drive assigned, or a number in the range 1..26 which is the last drive numer used to reference that drive (1 = A:, etc.). Thus, on a machine which has a single floppy drive, calling `ioctl (1, DOS_GLDRVMAP);` will return 2 if the floppy was last refered to as B:. This function and the next one can be used together to prevent DOS from popping the ugly prompt saying "Insert diskette for drive B: and press any key when ready".

`DOS_SLDRVMAP`
Set logical drive map. For example, a call like this:

```
ioctl (1, DOS_SLDRVMAP);
```

will cause drive A: to be mapped to drive B:.

`DOS_QGIOCTLCAPH`
Query generic ioctl capability (handle). Test if a handle supports ioctl functions beyond those in the standard DOS 3.2 set. Call like this:

```
int ret_val = ioctl (fd, DOS_QGIOCTLCAPH, category_and_function);
```

This will return zero if the specified IOCTL function is supported, 1 if not.

`DOS_QGIOCTLCAPD`
Query generic ioctl capability (drive). Test if a drive supports ioctl functions beyond those in the standard DOS 3.2 set. Used same as DOS_QGIOCTLCAPH, but the first argument is a drive number (0 = default, 1 = A:, etc.), not a handle.

If your specific device driver requires different commands, they must be or'ed together with the flags listed in `<sys/ioctl.h>` to tell the drive about transfer buffers and what to return.

Return Value

See description above.

Device information word

The bits of the device information word have the following meaning:\\ Character device:

14 Device can process IOCTL request

13 output until busy supported

11 driver supports OPEN/CLOSE calls

7 set (indicates device)

6 EOF on input

5 raw (binary) mode

4 device is special (uses INT 29)

3 clock device

2 NUL device

1 standard output

0 standard input

Disk file

15 file is remote (DOS 3.0+)

14 don't set file date/time on closing (DOS 3.0+)

11 media not removable

8 (DOS 4 only) generate INT 24 if no disk space on write or read past end of file

7 clear (indicates file)

6 file has not been written

5-0 drive number (0 = A:)

Example

```
#include <sys/ioctl.h>
int main(int argc, char **argv)
{

char buf[6];
short *s;

open(fd,"EMMQXXX0",O_RDONLY);
mybuf[0] = '\0';
s = mybuf;
ioctl(fd,DOS_SNDDATA,6, (int) &mybuf);
if(*s ==0x25 )printf("EMM386 >= 4.45\n");
mybuf[0]='\x02';
ioctl(fd,DOS_SNDDATA,2,(int )&mybuf);
printf("EMM Version %d.%d\n",(int )mybuf[0],(int) mybuf[1]);
close(fd);
}
```

ioctl (General description)

`ioctl` performs low level calls to communicate with device drivers. As there are lots of different device drivers, no really general description is possible.

The DJGPP version tries to cope with two different flavors of `ioctl`, a DOSish and a UNIXish way. To distinguish between DOS-like and UNIX-like calls, all valid DOS commands have all 3 MSB set to 0, the UNIX command have at least one of the 3 MSB set to 1.

ioctl (UNIX)

Syntax

```
#include <sys/ioctl.h>

int ioctl(int fd, int cmd, ... );
```

Description

The UNIX version first checks if an FSEXT handler is associated with the file descriptor fd. If so, it calls the handler in the usual way (See File System Extensions).

Otherwise,the operation specified by cmd is performed on the file open on handle fd. The following operations are defined by the header `sys/ioctl.h`:

TIOCGWINSZ

Fill in the winsize structure pointed to by the third argument with the screen width and height.

The winsize structure is defined in sys/ioctl.h as follows:

```
struct winsize
{

unsigned short ws_row; /* rows, in characters */
unsigned short ws_col; /* columns, in characters */
unsigned short ws_xpixel; /* horizontal size, pixels */
unsigned short ws_ypixel; /* vertical size, pixels */
};
```

Return Value

Zero for TIOCGWINSZ. Otherwise, -1 is returned and errno is set to ENOSYS for all others.

Example

```
#include <sys/ioctl.h>
#include <stdio.h>
int main(int argc, char **argv)
{

struct winsize sz;

ioctl(0, TIOCGWINSZ, &screen_size);
printf("Screen width: %i Screen height: %i\n", sz.ws_col, sz.ws_row);
return 0;
}
```

_is_cdrom_drive

Syntax

```
#include <dos.h>

int _is_cdrom_drive( const int drive );
```

Description

This function checks if drive number drive (1 == A:, 2 == B:, etc.) is a CD-ROM drive. It works with MSCDEX 2.x and Windows 9X built-in CDFS support.

Return Value

1 if the drive is a CDROM drive, otherwise 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{


if( _is_cdrom_drive( 'R' - 'A' + 1 ) )
{
printf("C: is a CDROM drive.\n");
}
else
{
printf("C: is not a CDROM drive.\n");
}
```

```
exit(0);
}
```

_is_DOS83

Syntax

```
#include <fcntl.h>

int _is_DOS83 (const char *fname);
```

Description

This function checks the filename pointed to by fname to determine if it is a standard short (8+3) form file name. The filename should only include the name part of the file. It is expected the filename will contain a valid long or short file name (no validation is done to exclude path characters or characters always illegal in any file name). Note: If the filename contains lower case characters the name is considered to be a long name and not a standard short name (and the function will return 0).

The function will return 0 (failure) if there are more than 8 characters before a period, more than 3 characters after a period, more than one period, starts with a period, any lower case characters, or any of the special characters `+ , ; = [ ]`, or a space. The special names `.` and `..` are exceptions and will return success.

This function could be called to determine if a filename is valid on DOS before long file name support. If this function returns 1 the filename probably does not have a long name entry on a FAT file system. The library internally calls this function to determine if a file should have its name lower cased when See _preserve_fncase is false.

Return value

The function returns an integer 0 (not DOS 8.3) or 1 (DOS 8.3)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_is_executable

Syntax

```
#include <sys/stat.h>

int _is_executable(const char *path, int fhandle,
const char *extension);
```

Description

This function determines if a file is executable under DOS/DJGPP environment. The file may be given either by its path or its file handle fhandle. If extension is non-NULL and non-empty, it is used first to look up in a list of known extensions which determine whether the file is executable. (If the `_STAT_EXEC_EXT` bit of the `_djstat_flags` global variable (See _djstat_flags) is not set, this step is skipped.) If extension is unavailable or not enough to determine the result, the first 2 bytes of the file are checked to contain one of the known magic numbers identifying the file as executable. If the file's 2 first bytes need to be read but the read fails, 0 is returned and errno is set. (The file is only searched for magic number if the `_STAT_EXEC_MAGIC` bit of the `_djstat_flags` variable is set.)

Note that if `_STAT_EXEC_MAGIC` is set, but `_STAT_EXEC_EXT` is not, some files which shouldn't be flagged as executables (e.g., COFF `*.o` object files) will have their execute bit set, because they have the magic number signature at their beginning. Therefore, only use the above combination if you want to debug the list of extensions provided in `is_exec.c` file from the library sources.

If the file passed by its handle was open as write-only, and the extension alone isn't enough to determine whether the file is executable, then this function returns 0, because it cannot look at the magic number.

This function is used internally by `f?stat`; you are not supposed to call it directly. However, if you call it, and pass file through path, it's up to you to resolve symlinks there.

Return Value

1 for executable file, 0 otherwise (including in case of errors in accessing the file).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_is_fat32

Syntax

```
#include <dos.h>

int _is_fat32( const int drive );
```

Description

This function checks if drive number drive (1 == A:, 2 == B:, etc.) is formatted with FAT32.

For performance reasons the result is cached, hence if a drive is reformatted either from or to FAT32 a DJGPP program must be restarted.

Return Value

1 if the drive is formatted with FAT32, otherwise 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{


if( _is_fat32( ’C’ - ’A’ + 1 ) )
{
printf("C: is a FAT32 drive.\n");
}
else
{
printf("C: is not a FAT32 drive.\n");
}

exit(0);
}
```

_is_ram_drive

Syntax

```
#include <dos.h>

int _is_ram_drive( const int drive );
```

Description

This function checks if drive number drive (1 == A:, 2 == B:, etc.) is a RAM disk. It is done by checking if the number of FAT copies (in the Device Parameter Block) is 1, which is typical of RAM disks. This doesn’t *have* to be so, but if it’s good enough for Andrew Schulman et al (Undocumented DOS, 2nd edition), we can use this as well.

Return Value

1 if the drive is a RAM drive, otherwise 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int i = 4;

printf("%c: is a RAM drive: %d.\n", 'A' - 1 + i, _is_ram_drive( i ) )
```

_is_remote_drive

Syntax

```
int _is_remote_drive(int drv);
```

Description

Given the drive number in drv (A: = 0, B: = 1, etc.), this function returns non-zero if the drive is treated by DOS as a remote (networked) drive, or zero otherwise. It does so by calling subfunction 09h of the DOS IOCTL function (interrupt 21h, AX=4409h) and looking at bit 12 of the device attribute word returned in the DX register.

Note that DOS treats CD-ROM drives as remote.

Return Value

Zero for local drives, non-zero for remote and CD-ROM drives.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_is_remote_handle

Syntax

```
int _is_remote_handle(int fhandle);
```

Description

Given the file handle of an open file in fhandle, this function returns non-zero if the drive where that file resides is treated by DOS as a remote (networked) drive, or zero otherwise. It does so by calling subfunction 0Ah of the DOS IOCTL function (interrupt 21h, AX=440Ah) and looking at bit 15 of the device attribute word returned in the DX register.

Note that DOS treats CD-ROM drives as remote.

Return Value

Zero for files on local drives, non-zero for files on remote and CD-ROM drives.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

isalnum

Syntax

```
#include <ctype.h>

int isalnum(int c);
```

Description

Tells if c is any letter or digit.

Return Value

Nonzero if c is a letter or digit, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isalpha

Syntax

```
#include <ctype.h>

int isalpha(int c);
```

Description

Tells if c is a letter.

Return Value

Nonzero if c is a letter, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isascii

Syntax

```
#include <ctype.h>

int isascii(int c);
```

Description

Tells if c is an ASCII character (0x00 to 0x7f).

Return Value

Nonzero if c is ASCII, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

isatty

Syntax

```
#include <unistd.h>

int isatty(int fd);
```

Description

Tells if the file descriptor refers to a terminal device or not.

Return Value

Nonzero if fd is a terminal device, zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
if (isatty(1))
fflush(stdout);
```

isblank

Syntax

```
#include <ctype.h>

int isblank(int c);
```

Description

Tells if c is a blank character. A blank character is used to separate words, e.g.: or `\t`.

Return Value

Nonzero if c is a blank character, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.2-1992; 1003.1-2001

iscntrl

Syntax

```
#include <ctype.h>

int iscntrl(int c);
```

Description

Tells if c is a control character.

Return Value

Nonzero if c is a control character, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isdigit

Syntax

```
#include <ctype.h>

int isdigit(int c);
```

Description

Tells if c is a digit.

Return Value

Nonzero if c is a digit, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isgraph

Syntax

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

```
int isgraph(int c);
```

Description
Tells if c is a visible printing character. Space is not included.

Return Value
Nonzero if c is a visible printing character, else zero.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

islower
Syntax
```
#include <ctype.h>

int islower(int c);
```

Description
Tells if c is lower case or not.

Return Value
Nonzero if c is lower case, else zero.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isprint
Syntax
```
#include <ctype.h>

int isprint(int c);
```

Description
Tells if c is a printing character, which includes the space character.

Return Value
Nonzero if c is a printing character, else zero.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

ispunct
Syntax
```
#include <ctype.h>

int ispunct(int c);
```

Description
Tells if c is any printing character except space and those indicated by `isalnum`.

Return Value

Nonzero if c is punctuation, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isspace

Syntax

```
#include <ctype.h>

int isspace(int c);
```

Description

Tells if c is whitespace, that is, carriage return, newline, form feed, tab, vertical tab, or space.

Return Value

Nonzero if c is whitespace, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isupper

Syntax

```
#include <ctype.h>

int isupper(int c);
```

Description

Tells if c is an upper case character or not.

Return Value

Nonzero if c is upper case, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

isxdigit

Syntax

```
#include <ctype.h>

int isxdigit(int c);
```

Description

Tells if c is a valid hexadecimal digit or not. This includes `[0-9a-fA-F]`.

Return Value

Nonzero if c is a hex digit, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

itoa

Syntax

```
#include <stdlib.h>

char * itoa(int value, char *string, int radix)
```

Description

This function converts its argument value into a null-terminated character string using radix as the base of the number system. The resulting string with a length of upto 33 bytes (including the optional sign and the terminating `NULL` is put into the buffer whose address is given by string. For radixes other than 10, value is treated as an unsigned int (i.e., the sign bit is not interpreted as such). The argument radix should specify the base, between 2 and 36, in which the string reprsentation of value is requested.

Return Value

A pointer to string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char binary_str[33];

(void)itoa(num, binary_str, 2);
```

kbhit

Syntax

```
#include <pc.h>

int kbhit(void);
```

Description

If the user has hit a key, this function will detect it. This function is very fast when there is no key waiting, so it may be used inside loops as needed.

If you test shift/alt/ctrl status with bios calls (e.g., using `bioskey (2)` or `bioskey (0x12)`) then you should also use bios calls for testing for keys. This can be done with by `bioskey (1)` or `bioskey (0x11)`. Failing to do so can cause trouble in multitasking environments like DESQview/X.

Return Value

Nonzero if a key has been hit, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
while (!kbhit())
do_stuff();
```

kill

Syntax

```
#include <signal.h>

int kill(pid_t _pid, int _sig);
```

Description

If _pid is the current `getpid()`, the given _sig is raised with See raise.

Return Value

-1 on error, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

l64a

Syntax

```
#include <stdlib.h>

char *l64a(long value);
```

Description

This function takes a `long` argument and returns a pointer to its radix-64 representation. Negative values are supported as well. The resulting string can be turned back into a `long` value by the `a64l` function (See a64l).

Return Value

A pointer to a static buffer containing the radix-64 representation of value. Subsequent calls will overwrite the contents of this buffer. If value is `0L`, this function returns an empty string.

Radix-64

The radix-64 ASCII representation is a notation whereby 32-bit integers are represented by up to 6 ASCII characters; each character represents a single radix-64 digit. Radix-64 refers to the fact that each digit in this representation can take 64 different values. If the `long` type is more than 32 bits in size, only the low-order 32 bits are used. The characters used to represent digits are `.` (dot) for 0, `/` for 1, `0` through `9` for 2 to 11, `A` through `Z` for 12 to 37, and `a` through `z` for 38 to 63.

Note that this is *not* the same encoding used by either uuencode or the MIME base64 encoding.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. This function is new to the Posix 1003.1-200x draft

labs

Syntax

```
#include <stdlib.h>

long labs(long x);
```

Description

This function takes the absolute value of x. See abs.

Return Value

|x|

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

lchown

Syntax

```
#include <unistd.h>
```

```
int lchown(const char *file, int owner, int group);
```

Description

This function does nothing under MS-DOS.

Return Value

This function always returns zero if the file exists (it does not follow symbolic links), else it returns -1 and sets `errno` to `ENOENT`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ldexp

Syntax

```
#include <math.h>

double ldexp(double val, int exp);
```

Description

This function computes val*2^exp.

Return Value

val*2^exp. `ldexp(0., exp)` returns 0 for all values of `exp`, without setting `errno`. For non-zero values of val, `errno` is set to `ERANGE` if the result cannot be accurately represented by a `double`, and the return value is then the nearest representable `double` (possibly, an `Inf`). If val is a `NaN` or `Inf`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
ldexp(3.5,4) == 3.5 * (2^4) == 56.0
```

ldiv

Syntax

```
#include <stdlib.h>

ldiv_t ldiv(long numerator, long denominator);
```

Description

Returns the quotient and remainder of the division numerator divided by denominator. The return type is as follows:

```
typedef struct {
long quot;
long rem;
} ldiv_t;
```

Return Value

The results of the division are returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
ldiv_t l = ldiv(42, 3);
printf("42 = %ld x 3 + %ld\n", l.quot, l.rem);

ldiv(+40, +3) = { +13, +1 }
ldiv(+40, -3) = { -13, +1 }
ldiv(-40, +3) = { -13, -1 }
ldiv(-40, -3) = { +13, -1 }
```

lfilelength

Syntax

```
#include <io.h>

long long lfilelength(int fhandle);
```

Description

This function returns the size, in bytes, of a file whose handle is specified in the argument fhandle. To get the handle of a file opened by `fopen` (See fopen) or `freopen` (See freopen), you can use `fileno` macro (See fileno).

Return Value

The size of the file in bytes, or (if any error occured) -1LL and `errno` set to a value describing the cause of the failure.

The return value is of type `long long` which allows file sizes of 2^63-1 bytes to be returned. Note that FAT16 limits files to near 2^31 bytes and FAT32 limits files to 2^32-2 bytes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("Size of file to which STDIN is redirected is %ld\n",
lfilelength(0));
```

_lfn_gen_short_fname

Syntax

```
#include <fcntl.h>

char * _lfn_gen_short_fname (const char *long_fname, char *short_fname);
```

Description

This function generates a short (8+3) filename alias for the long filename pointed to by long_fname and puts it into the buffer pointed to by short_fname. It uses the same algorithm that Windows 9x uses, with the exception that the returned short name will never have a numeric tail, because this function doesn't check the directory to see whether the generated short name will collide with any other file in the directory. Note that long_fname must contain only the name part of a file; elements of a full pathname (like **:** or **/** are not allowed (they will cause the function to fail). short_fname will be returned upper-cased, since that is how 8+3 filenames are stored in directory entries.

When the LFN API is not supported (See _use_lfn), the function simply converts up to 12 characters of long_fname to upper-case and returns that. It will do the same if long_fname includes any characters illegal in a filename.

This function returns incorrect results on Windows 2000 and Windows XP due to bugs in the implementation of the DPMI call on those platforms. Do not use this function in those environments.

You might need to call this function if you want to know whether a given filename is valid on MSDOS: if a case-sensitive string comparison function such as `strcmp` (See strcmp) returns a 0 when it compares the original long filename with the short one returned by `_lfn_gen_short_fname`, then the filename is a valid DOS name. (Note that if long_fname is in lower case, it might not compare equal with short_fname because of the case difference.)

Return value

The function returns a pointer to short_fname.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int dos_check (char *fname)
{

char fshort[13];
int retval;

if (stricmp (_lfn_gen_short_fname (fname, fshort), fname) == 0)
{
printf ("%s is a valid MSDOS 8+3 filename\n", fname);
retval = 1;
}
else
{
printf ("%s will have to be changed for MSDOS\n", fname);
retval = 0;
}
return retval;
}
```

_lfn_get_ftime

Syntax

```
#include <fcntl.h>

char _lfn_get_ftime (int fhandle, int flag);
```

Description

This function returns creation and access time for files that reside on a filesystem which supports long filenames (such as Windows 95). Files which reside on native FAT filesystems will cause this function to fail. The fhandle parameter is the file handle as returned by one of the functions which open or create files. The flag parameter determines which time (creation or access) is returned. It can be set to one of the following:

`_LFN_ATIME`
: Causes `_lfn_get_ftime` to return the time when the file was last accessed. (Currently, it actually only returns the *date* of last access; the time bits are all zeroed.)

`_LFN_CTIME`
: Causes `_lfn_get_ftime` to return the time when the file was created. Note that if the file was created by a program which doesn't support long filenames, this time will be zero.

Return value

The file time stamp, as a packed unsigned int value:

`Bits 0-4`
: seconds divided by 2

`Bits 5-10`
: minutes (0-59)

`Bits 11-15`
: hours (0-23)

`Bits 16-20`

day of the month (1-31)

`Bits 21-24`
month (1 = January)

`Bits 25-31`
year offset from 1980 (add 1980 to get the actual year)

If the underlying system calls fail, the function will return 0 and set `errno` to an appropriate value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned file_stamp = _lfn_get_ftime (handle, _LFN_CTIME);
```

__libc_termios_exist_queue

Syntax

```
#include <libc/ttyprvt.h>

int __libc_termios_exist_queue (void);
```

Description

This function checks whether there are any characters buffered in the termios internal queue. Functions that work on file handles which might be hooked by termios should call this function before they invoke system calls which test if there are any characters available for input.

Return Value

Non-zero if some characters are buffered, zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__libc_termios_init

Syntax

```
#include <libc/ttyprvt.h>

void __libc_termios_init (void);
```

Description

This function sets read/write hooks for the termios emulation and import parameters. Currently importing parameters is not supported, the emulation is resolved by only internal(static) parameters. Note that this function is automatically called by the other termios functions.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

libm

Syntax

```
#include <libm/math.h>

enum fdversion {fdlibm_ieee = -1, fdlibm_svid, fdlibm_xopen,
fdlibm_posix};

#define _LIB_VERSION_TYPE enum fdversion
#define _LIB_VERSION _fdlib_version

extern _LIB_VERSION_TYPE _LIB_VERSION;
```

```
#define _IEEE_ fdlibm_ieee
#define _SVID_ fdlibm_svid
#define _XOPEN_ fdlibm_xopen
#define _POSIX_ fdlibm_posix

_LIB_VERSION_TYPE _LIB_VERSION = _IEEE_;
```

Description

The alternate math library, `libm.a`, originally written by Cygnus support, provides versions of mathematical functions which comply to several different standards of behavior in abnormal cases, and are sometimes more accurate than those included in the default `libc.a` library, in particular when elaborate argument reduction is required to avoid precision loss. Functions in `libm.a` allow to create programs with well-defined and standard-compliant behavior when numerical errors occur, and provide the application with a means to control their behavior in abnormal cases via the `matherr` callback. They almost never rely on the features specific to the x87 FPU, and are thus slower and sometimes slightly less accurate than the functions from `libc.a`.

In contrast, the functions in the default `libc.a` library are written for maximum speed and exploitation of the x87 FPU features, do not call `matherr`, and are therefore much faster and sometimes more accurate (due to the extended 80-bit precision with which the x87 FPU carries its calculations).

Another aspect of differences between functions in `libc.a` and in `libm.a` is the value returned when the result overflows a `double`. The functions from `libc.a` always return a suitably signed infinity, `Inf`, whereas for functions from `libm.a` an application can arrange for a large but finite value to be returned. Getting finite return values might be important in certain kinds of mathematical computations where the special rules defined for infinities (e.g., Inf + a = Inf) might be inappropriate.

Refer to See Math, description of the `libm.a` functions, Mathematical Functions, libm, The Cygnus C Math Library, for detailed documentation of the individual functions from `libm.a`. This section explains the general setup of using those functions from DJGPP programs.

To use the alternate math library with your program, you need to do the following:

- Include the header `<libm/math.h>`. Alternatively, you can include `<math.h>` as usual and compile with `-D_USE_LIBM_MATH_H` option to `gcc`, which will cause it to use `libm/math.h` instead of the default `math.h`. (The second possibility leaves the source ANSI-compliant.)

- Set the global variable `_fdlib_version` to a value other than the default `_IEEE_`. The possible values are listed and explained below.

- At the beginning of your `main` function, set the FPU to a predictable state by calling `_clear87` (See _clear87) and `_fpreset` (See _fpreset) library functions. (Another possibility is to make these calls in a function declared with `__attribute__((constructor))`, so it will be called before `main`.)

- Link your program with the `libm.a` library, e.g. by specifying `-lm` on the link command line.

The functions in `libm.a` can emulate different standards. You can select to which standard your program will comply by setting the global variable `_fdlib_version` (or the macro `_LIB_VERSION` which evaluates to it) to one of the values below. This will only affect the behavior of the math functions when an error is signaled by the FPU.

`_IEEE_`
: The default value, specifies IEEE-compliant operation. In case of an error, this version will immediately return whatever result is computed by the FPU, and will **not** set `errno`. If the result overflows, an `Inf` is returned. This version gives the fastest code.

`_POSIX_`
: In case of an error, this version will set `errno` to the appropriate value (`EDOM` or `ERANGE`) and return to the caller, without calling the `matherr` function (See matherr). If the result overflows, an `Inf` is returned. This version should be used for maximum POSIX- and ANSI-compliance.

`_SVID_`
: This version is compliant with the System V Interface Definition. This is the slowest version. In case of an error, it calls the `matherr` function (See matherr), which can be customized to the specific application needs. If `matherr` returns zero, a message is printed to the standard error stream which states the name of the function that generated the error and the error type, and `errno` is set. If `matherr` returns non-zero, there will be no message and `errno` will be left unaltered. If the result overflows, this version returns `HUGE`, a large but finite value defined by `libm/math.h`.

`_XOPEN_`
: Complies to the X/Open specifications. It behaves exactly like `_SVID_`, but it never prints an error message, even if `matherr` returns zero, and `Inf` us returned when a result overflows.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Testing errno == EDOM after sqrt(-1).

!!! MUST compile with -lm !!! */

#include <assert.h>
#include <errno.h>
#include <stdio.h>
#include <libm/math.h> /* or #define _USE_LIBM_MATH_H
* and #include <math.h> */
#include <float.h>

/* Setting _LIB_VERSION to anything but _IEEE_ will turn on
* errno handling. */
_LIB_VERSION_TYPE _LIB_VERSION = _POSIX_;

int main (void)
{

/* Reset the FPU (possible previous FP problems). */
_clear87 ();
_fpreset ();

/* Run the test. */
errno = 0;
assert(errno == 0);
sqrt(-1.0);
assert(errno == EDOM); /* this line should NOT cause
* the assertion to fail */

return(0);
}
```

link

Syntax

```
#include <unistd.h>

int link(const char *exists, const char *new);
```

Description

Because of limitations of MS-DOS, this function doesn't really link two files together. However, it simulates a real `link` by copying the file at exists to new.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
link("foo.c", "foo.bak");
```

llabs

Syntax

```
#include <stdlib.h>

long long int llabs(long long x);
```

Description

This function takes the absolute value of x. See abs.

Return Value

|x|

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

lldiv

Syntax

```
#include <stdlib.h>

lldiv_t lldiv(long long int numerator, long long int denominator);
```

Description

Returns the quotient and remainder of the division numerator divided by denominator. The return type is as follows:

```
typedef struct {
long long int quot;
long long int rem;
} lldiv_t;
```

Return Value

The results of the division are returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
lldiv_t l = lldiv(42, 3);
printf("42 = %lld x 3 + %lld\n", l.quot, l.rem);

lldiv(+40, +3) = { +13, +1 }
lldiv(+40, -3) = { -13, +1 }
lldiv(-40, +3) = { -13, -1 }
lldiv(-40, -3) = { +13, -1 }
```

llockf

Syntax

```
#include <unistd.h>

int llockf (int fildes, int function, offset_t size);
```

Description

The `llockf` function is a simplified interface to the locking facilities of `fcntl` (see See fcntl, for more detailed information).

The `llockf` function performs exactly the same functions as `lockf` (See lockf), with exactly the same input commands and results, but the input size parameter is of type `offset_t` instead of `off_t`, and the `fcntl` calls are made using functions `F_GETLK64`, `F_SETLK64` and `F_SETLKW64`, where `lockf` uses functions `F_GETLK`, `F_SETLK` and `F_SETLKW`.

The `llockf` function is intended to permit using `fcntl` functions with size values greater than 2^31 - 1.

fildes is an open file descriptor.

function is a control value which specifies the action to be taken. The permissible values for function are defined in <unistd.h> as follows:

```
#define F_ULOCK 0 /* Unlock a previously locked section */
#define F_LOCK 1 /* Lock a section for exclusive use */
#define F_TLOCK 2 /* Test and lock a section for exclusive use */
#define F_TEST 3 /* Test section for other locks */
```

All other values of function are reserved for future extensions and will result in an error return if not implemented, with `errno` set to `EINVAL`.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If size is zero, the section from the current offset through the largest file offset is locked (i.e. from the current offset through the end of file).

The functions defined for `llockf` are as follows:

`F_TEST`
: This function is used to detect if the specified section is already locked.

`F_LOCK`
`F_TLOCK`
: `F_LOCK` and `F_TLOCK` both lock a section of a file, if the section is available. These two function requests differ only by the action taken if the resource is not available. `F_LOCK` will cause the calling program to wait until the resource is available. `F_TLOCK` will cause the function to return a -1 and set `errno` to `EACCES` if the section is already locked.

`F_ULOCK`
: `F_ULOCK` removes locks from a section of the file. This function will release locked sections controlled by the program.

The `llockf` function will fail, returning -1 and setting `errno` to the following error values if the associated condition is true:

`EBADF`
: Parameter fildes is not a valid open file.

`EACCES`
: Parameter function is `F_TLOCK` or `F_TEST` and the section is already locked.

`EINVAL`
: Parameter function is not one of the implemented functions. Or: An attempt was made to lock a directory, which is not supported.

All lock requests in this implementation are coded as exclusive locks (i.e. all locks use the `fcntl` request `F_WRLCK`).

It is therefore wise to code `llockf` by using function `F_TLOCK` with all lock requests, and to test the return value to determine if the lock was obtained or not. Using `F_TLOCK` will cause the implementation to use `F_SETLK` instead of `F_SETLKW`, which will return an error if the lock cannot be obtained.

Return Value

On success, zero is returned. On error, -1 is returned, and `errno` is set appropriately, as described above.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Request a lock on file handle fd from the current position to
the end of the file */
errno = 0;
retval = llockf(fd, F_LOCK, 0);

/* Request a non-blocking lock on file handle fd */
errno = 0;
retval = llockf(fd, F_TLOCK, 0);

/* Test a lock on file handle fd */
errno = 0;
retval = llockf(fd, F_TEST, 0);

/* Release a lock on file handle fd */
errno = 0;
retval = llockf(fd, F_ULOCK, 0);
```

llseek

Syntax

```
#include <unistd.h>

offset_t llseek(int fd, offset_t offset, int whence);
```

Description

This function moves the file pointer for fd according to whence:

`SEEK_SET`
: The file pointer is moved to the offset specified.

`SEEK_CUR`
: The file pointer is moved relative to its current position.

`SEEK_END`
: The file pointer is moved to a position offset bytes from the end of the file. The offset is usually nonpositive in this case.

offset is of type long long, thus enabling you to seek with offsets as large as ~2^63 (FAT16 limits this to ~2^31; FAT32 limits this to 2^32-2).

Return Value

The new offset is returned. Note that due to limitations in the underlying DOS implementation the offset wraps around to 0 at offset 2^32. -1 means the call failed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
long long ret;

ret = llseek(fd, (1<<32), SEEK_SET); /* Now ret equals 0
* (unfortunately). */
ret = llseek(fd, -1, SEEK_CUR); /* Now ret equals 2^32-1 (good!). */
ret = llseek(fd, 0, SEEK_SET); /* Now ret equals 0 (good!). */
ret = llseek(fd, -1, SEEK_CUR); /* Now ret equals 2^32-1 (bad). */
```

load_npx

Syntax

```
#include <debug/dbgcom.h>

extern NPX npx;
```

```
void load_npx (void);
```

Description

This function restores the state of the x87 numeric processor from the data saved in the external variable `npx`. This variable is a structure defined as follows in the header `debug/dbgcom.h`:

```
typedef struct {
unsigned short sig0;
unsigned short sig1;
unsigned short sig2;
unsigned short sig3;
unsigned short exponent:15;
unsigned short sign:1;
} NPXREG;

typedef struct {
unsigned long control;
unsigned long status;
unsigned long tag;
unsigned long eip;
unsigned long cs;
unsigned long dataptr;
unsigned long datasel;
NPXREG reg[8];
long double st[8];
char st_valid[8];
long double mmx[8];
char in_mmx_mode;
char top;
} NPX;
```

`load_npx` should be called immediately after `run_child` (See run_child) is called to begin or resume the debugged program, and provided that a call to `save_npx` was issued before `run_child` was called. See save_npx.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
save_npx ();
run_child ();
load_npx ();
```

localeconv

Syntax

```
#include <locale.h>

struct lconv *localeconv(void);
```

Description

This function returns a pointer to a static structure that contains information about the current locale. The structure contains these fields:

`char *currency_symbol`
: A string that should be used when printing local currency.

`char *decimal_point`
: A string that is used to separate the integer and fractional portions of real numbers in `printf`. Currently, only the first character is significant.

`char *grouping`
: An array of numbers indicating the size of groupings for non-monetary values to the left of the decimal point. The first number is the size of the grouping just before the decimal point. A number of zero means to repeat the previous number indefinitely. A number of `CHAR_MAX` means to group the remainder of the

digits together.

`char *int_curr_symbol`
: A string that should be used when formatting monetary values for local currency when the result will be used internationally.

`char *mon_decimal_point`
: A string that separates the interger and fractional parts of monetary values.

`char *mon_grouping`
: Same as grouping, but for monetary values.

`char *negative_sign`
: A string that is used to represent negative monetary values.

`char *positive_sign`
: A string that is used to represent positive monetary values.

`char *thousands_sep`
: The grouping separator for non-monetary values.

`char frac_digits`
: The number of digits to the right of the decimal point for monetary values.

`char int_frac_digits`
: Like frac_digits, but when formatting for international use.

`char n_cs_precedes`
: If nonzero, the currency string should precede the monetary value if the monetary value is negative.

`char n_sep_by_space`
: If nonzero, the currency string and the monetary value should be separated by a space if the monetary value is negative.

`char n_sign_posn`
: Determines the placement of the negative indication string if the monetary value is negative.

0
: ($value), (value$)

1
: -$value, -value$

2
: $value-, value$-

3
: -$value, value-$

4
: $-value, value$-

`char p_cs_precedes`
`char p_sep_by_space`
`char p_sign_posn`
: These are the same as n_*, but for when the monetary value is positive.

Note that any numeric field may have a value of `CHAR_MAX`, which indicates that no information is available.

Return Value

A pointer to the `struct lconv` structure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
struct lconv *l = localeconv;
printf("%s%d\n", l->negative_sign, value);
```

localtime

Syntax

```
#include <time.h>

struct tm *localtime(const time_t *tod);
```

Description

Converts the time represented by tod into a structure, correcting for the local timezone. See See gmtime, for the description of `struct tm`.

Return Value

A pointer to a static structure which is overwritten with each call.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

lock

Syntax

```
#include <io.h>

int lock(int fd, long offset, long length);
```

Description

Locks a region in file fd using MS-DOS file sharing interface. The region of length bytes, starting from offset, will become entirely inaccessible to other processes. If multiple locks are used on a single file they must be non-overlapping. The lock must be removed before the file is closed.

This function will fail unless `share`, or a network software providing similar interface, is installed. This function is compatible with Borland C

++

function of the same name.

See unlock.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

lock64

Syntax

```
#include <io.h>

int lock64(int fd, long long offset, long long length);
```

Description

Locks a region in file fd using MS-DOS file sharing interface. The region of length bytes, starting from offset, will become entirely inaccessible to other processes. If multiple locks are used on a single file they must be non-overlapping. The lock must be removed before the file is closed.

This function will fail unless SHARE, or a network software providing similar interface, is installed.

Arguments offset and length must be of type `long long`, thus enabling you to lock with offsets and lengths as large as ~2^63 (FAT16 limits this to ~2^31; FAT32 limits this to 2^32-2).

See unlock64.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

lockf

Syntax

```
#include <unistd.h>

int lockf (int fildes, int function, off_t size);
```

Description

The `lockf` function is a simplified interface to the locking facilities of `fcntl` (see See fcntl, for more detailed information).

fildes is an open file descriptor.

function is a control value which specifies the action to be taken. The permissible values for function are defined in <unistd.h> as follows:

```
#define F_ULOCK 0 /* Unlock a previously locked section */
#define F_LOCK 1 /* Lock a section for exclusive use */
#define F_TLOCK 2 /* Test and lock a section for exclusive use */
#define F_TEST 3 /* Test section for other locks */
```

All other values of function are reserved for future extensions and will result in an error return if not implemented, with `errno` set to `EINVAL`.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If size is zero, the section from the current offset through the largest file offset is locked (i.e. from the current offset through the end of file).

The functions defined for `lockf` are as follows:

`F_TEST`
: This function is used to detect if the specified section is already locked.

`F_LOCK`
`F_TLOCK`
: `F_LOCK` and `F_TLOCK` both lock a section of a file, if the section is available. These two function requests

differ only by the action taken if the resource is not available. F_LOCK will cause the calling program to wait until the resource is available. F_TLOCK will cause the function to return a -1 and set errno to EACCES if the section is already locked.

F_ULOCK
: F_ULOCK removes locks from a section of the file. This function will release locked sections controlled by the program.

The lockf function will fail, returning -1 and setting errno to the following error values if the associated condition is true:

EBADF
: Parameter fildes is not a valid open file.

EACCES
: Parameter function is F_TLOCK or F_TEST and the section is already locked.

EINVAL
: Parameter function is not one of the implemented functions. Or: An attempt was made to lock a directory, which is not supported.

All lock requests in this implementation are coded as exclusive locks (i.e. all locks use the fcntl request F_WRLCK).

It is therefore wise to code lockf by using function F_TLOCK with all lock requests, and to test the return value to determine if the lock was obtained or not. Using F_TLOCK will cause the implementation to use F_SETLK instead of F_SETLKW, which will return an error if the lock cannot be obtained.

Return Value

On success, zero is returned. On error, -1 is returned, and errno is set appropriately, as described above.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Request a lock on file handle fd from the current position to
the end of the file */
errno = 0;
retval = lockf(fd, F_LOCK, 0);

/* Request a non-blocking lock on file handle fd */
errno = 0;
retval = lockf(fd, F_TLOCK, 0);

/* Test a lock on file handle fd */
errno = 0;
retval = lockf(fd, F_TEST, 0);

/* Release a lock on file handle fd */
errno = 0;
retval = lockf(fd, F_ULOCK, 0);
```

log

Syntax

```
#include <math.h>

double log(double x);
```

Description

This function computes the natural logarithm of x.

Return Value

The natural logarithm of x. If x is zero, a negative infinity is returned and errno is set to ERANGE. If x is negative or `+Inf` or a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

log10

Syntax

```
#include <math.h>

double log10(double x);
```

Description

This function computes the base-10 logarithm of x.

Return Value

The logarithm base 10 of x. If x is zero, a negative infinity is returned and `errno` is set to `ERANGE`. If x is negative or `+Inf` or a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

log1p

Syntax

```
#include <math.h>

double log1p(double x);
```

Description

This function computes the natural logarithm of 1 + x. It is more accurate than `log(1 + x)` for small values of x.

Return Value

The natural logarithm of 1 + x. If x is -1, a negative infinity is returned and `errno` is set to `ERANGE`. If x is less than -1 or `+Inf` or a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

log2

Syntax

```
#include <math.h>

double log2(double x);
```

Description

This function computes the base-2 logarithm of x.

Return Value

The base-2 logarithm of x. If x is zero, a negative infinity is returned and `errno` is set to `ERANGE`. If x is negative or `+Inf` or a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

longjmp

Syntax

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

Description

This function reverts back to a CPU state that was stored in env by `setjmp` (See setjmp). The state includes all CPU registers, so any variable in a register when `setjmp` was called will be preserved, and all else will be indeterminate.

The value passed as val will be the return value of `setjmp` when it resumes processing there. If val is zero, the return value will be one.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
jmp_buf j;
if (setjmp(j))
return;
do_something();
longjmp(j, 1);
```

lowvideo

Syntax

```
#include <conio.h>

void lowvideo(void);
```

Description

Causes any new characters put on the screen to be dim.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

lseek

Syntax

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Description

This function moves the file pointer for handle fd according to whence:

`SEEK_SET`
: The file pointer is moved to the offset specified.

`SEEK_CUR`
: The file pointer is moved offset bytes relative to its current position.

`SEEK_END`
: The file pointer is moved to a position offset bytes from the end of the file. The value of offset is usually nonpositive in this case.

Return Value

The new offset is returned, or -1 and `errno` set on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
lseek(fd, 12, SEEK_CUR); /* skip 12 bytes */
```

lstat

Syntax

```
#include <sys/stat.h>

int lstat(const char *file, struct stat *sbuf);
```

Description

This function obtains the status of the file file and stores it in sbuf. If file is a symbolic link, then `lstat` returns information about the symbolic link. To get information about the target of a symbolic link, use `stat` (See stat) instead.

sbuf has this structure:

```
struct stat {
time_t st_atime; /* time of last access */
time_t st_ctime; /* time of file's creation */
dev_t st_dev; /* The drive number (0 = a:) */
gid_t st_gid; /* what getgid() returns */
ino_t st_ino; /* starting cluster or unique identifier */
mode_t st_mode; /* file mode - S_IF* and S_IRUSR/S_IWUSR */
time_t st_mtime; /* time that the file was last written */
nlink_t st_nlink; /* 2 + number of subdirs, or 1 for files */
off_t st_size; /* size of file in bytes */
blksize_t st_blksize; /* block size in bytes*/
uid_t st_uid; /* what getuid() returns */
dev_t st_rdev; /* The drive number (0 = a:) */
};
```

The `st_atime`, `st_ctime` and `st_mtime` have different values only when long file names are supported (e.g. on Windows 9X); otherwise, they all have the same value: the time that the file was last written *Even when long file names are supported, the three time values returned by* `lstat` *might be identical if the file was last written by a program which used legacy DOS functions that don't know about long file names.*. Most Windows 9X VFAT filesystems only support the date of the file's last access (the time is set to zero); therefore, the DJGPP implementation of `lstat` sets the `st_atime` member to the same value as `st_mtime` if the time part of `st_atime` returned by the filesystem is zero (to prevent the situation where the file appears to have been created *after* it was last accessed, which doesn't look good).

The `st_size` member is an signed 32-bit integer type, so it will overflow on FAT32 volumes for files that are larger than 2GB. Therefore, if your program needs to support large files, you should treat the value of `st_size` as an unsigned value.

For some drives `st_blksize` has a default value, to improve performance. The floppy drives A: and B: default to a block size of 512 bytes. Network drives default to a block size of 4096 bytes.

Some members of `struct stat` are very expensive to compute. If your application is a heavy user of `lstat` and is too slow, you can disable computation of the members your application doesn't need, as described in See _djstat_flags.

Return Value

Zero on success, nonzero on failure (and `errno` set).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct stat s;
lstat("data.txt", &s);
if (S_ISDIR(s.st_mode))
printf("is directory\n");
```

Implementation Notes

Supplying a 100% Unix-compatible `lstat` function under DOS is an implementation nightmare. The following notes describe some of the obscure points specific to `lstat`s behavior in DJGPP.

1. The `drive` for character devices (like `con`, `/dev/null` and others is returned as -1. For drives networked by Novell Netware, it is returned as -2.

2. The starting cluster number of a file serves as its inode number. For files whose starting cluster number is inaccessible (empty files, files on Windows 9X, on networked drives, etc.) the `st_inode` field will be *invented* in a way which guarantees that no two different files will get the same inode number (thus it is unique). This invented inode will also be different from any real cluster number of any local file. However, only on plain DOS, and only for local, non-empty files/directories the inode is guaranteed to be consistent between `stat`, `fstat` and `lstat` function calls. (Note that two files whose names are identical but for the drive letter, will get the same invented inode, since each filesystem has its own independent inode numbering, and comparing files for identity should include the value of `st_dev`.)

3. The WRITE access mode bit is set only for the user (unless the file is read-only, hidden or system). EXECUTE bit is set for directories, files which can be executed from the DOS prompt (batch files, .com, .dll and .exe executables) or run by `go32-v2`.

4. Size of directories is reported as the number of its files (sans '.' and '..' entries) multiplied by 32 bytes (the size of directory entry). On FAT filesystems that support the LFN API (such as Windows 9X), the reported size of the directory accounts for additional space used to store the long file names.

5. Time stamp for root directories is taken from the volume label entry, if that's available; otherwise, it is reported as 1-Jan-1980.

6. The variable `_djstat_flags` (See _djstat_flags) controls what hard-to-get fields of `struct stat` are needed by the application.

7. `lstat` should not be used to get an up-to-date info about a file which is open and has been written to, because `lstat` will only return correct data after the file is closed. Use `fstat` (See fstat) while the file is open. Alternatively, you can call `fflush` and `fsync` to make the OS flush all the file's data to the disk, before calling `lstat`.

8. The number of links `st_nlink` is always 1 for files other than directories. For directories, it is the number of subdirectories plus 2. This is so that programs written for Unix that depend on this to optimize recursive traversal of the directory tree, will still work.

mallinfo

Syntax

```
#include <stdlib.h>

struct mallinfo mallinfo(void);
```

Description

This function returns information about heap space usage. It is intended to be used for debugging dynamic memory allocation and tracking heap usage. The `struct mallinfo` structure is defined by `stdlib.h` as follows:

```
struct mallinfo {
int arena;
int ordblks;
int smblks;
int hblks;
int hblkhd;
int usmblks;
int fsmblks;
int uordblks;
int fordblks;
int keepcost;
};
```

whose members are:

`arena`
: The total amount of space, in bytes, handed by `sbrk` to `malloc`. Note that this is not the same as `sbrk(0)`, since `sbrk` allocates memory in large chunks and then subdivides them and passes them to `malloc` as required. In particular, the result of `sbrk(0)` might be much larger than the `arena` member of `struct mallinfo` when the DPMI host allocates memory in non-contiguous regions (happens on MS-Windows).

`ordblks`
: The number of ''ordinary blocks'': the total number of allocated and free blocks maintained by `malloc`.

`smblks`
: The number of ''small blocks''. This is normally zero, unless `malloc` was compiled with the symbol `NUMSMALL` defined to a non-zero value. Doing so activates an optional algorithm which serves small allocations quickly from a special pool. If this option is activated, the `smblks` member returns the number of free small blocks (the allocated small blocks are included in the value of `ordblks`).

`hblks`
`hblkhd`
: Always zero, kept for compatibility with other systems.

`usmblks`
: The space (in bytes) in ''small blocks'' that are in use. This is always zero in the DJGPP implementation.

`fsmblks`
: The space in free ''small blocks''. Non-zero only of `malloc` was compiled with `NUMSMALL` defined to a non-zero value. In that case, gives the amount of space in bytes in free small blocks.

`uordblks`
: The amount of space, in bytes, in the heap space currently used by the application. This does not include the small overhead (8 bytes per block) used by `malloc` to maintain its hidden information in each allocated block.

`fordblks`
: The amount of free heap space maintained by `malloc` in its free list.

`keepcost`
: Always zero, kept for compatibility.

Return Value

The `mallinfo` structure filled with information.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No (see note 1)

Notes:

1. This function is available on many Unix systems.

Example

```
struct mallinfo info = mallinfo();

printf("Memory in use: %d bytes\n",
info.usmblks + info.uordblks);
printf("Total heap size: %d bytes\n", info.arena);
```

malloc

Syntax

```
#include <stdlib.h>

void *malloc(size_t size);
```

Description

This function allocates a chunk of memory from the heap large enough to hold any object that is size bytes in length. This memory must be returned to the heap with `free` (See free).

Note: this version of malloc is designed to reduce memory usage. A faster but less efficient version is available in the libc sources (`djlsr*.zip`) in the file `src/libc/ansi/stdlib/fmalloc.c`.

Return Value

A pointer to the allocated memory, or `NULL` if there isn't enough free memory to satisfy the request.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char *c = (char *)malloc(100);
```

malloc hook functions

Syntax

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

void (*__libc_malloc_hook)(size_t size, void *block);
void (*__libc_malloc_fail_hook)(size_t size);
void (*__libc_free_hook)(void *block);
void (*__libc_free_null_hook)(void);
void (*__libc_realloc_hook)(void *block, size_t size);
```

Description

These hooks are provided for building custom `malloc` debugging packages. Such packages typically need to be notified when memory is allocated and freed by the application, in order to be able to find memory leaks, code that writes beyond the limits of allocated buffers or attempts to free buffers which were not allocated by `malloc`, etc. These hooks can be used to define callback functions which will be called by the library at strategic points. Each callback is only called if it is non-`NULL`; by default, all of them are initialized to a `NULL` value.

`__libc_malloc_hook`
: Called just before a chunk of memory is about to be returned to the application in response to an allocation request. size is the size requested by the application (**not** the actual size of the allocated buffer, which may be larger). block is a pointer to the block that was allocated, which is 4 bytes before the pointer that `malloc` will return to the application; these 4 bytes are used to record the actual size of the buffer. An additional copy of the block's size is recorded immediately after the buffer's end. Thus, `*(size_t *)((char *)block + 4 + (BLOCK *)block->size)` gives the second copy of the block's size.

`__libc_malloc_fail_hook`
: Called if `malloc` failed to find a free block large enough to satisfy a request, and also failed to obtain additional memory from `sbrk`. size is the requested allocation size.

`__libc_free_hook`
: Called when a buffer is about to be freed. block is a pointer 4 bytes before the address passed to `free` by the application, i.e. it is a pointer to the beginning of the size information maintained before the user buffer.

`__libc_free_null_hook`
: Called whenever a `NULL` pointer is passed to `free`. ANSI C specifically rules that this is allowed and should have no effect, but you might want to catch such cases if your program needs to be portable to old compilers whose libraries don't like `NULL` pointers in `free`.

`__libc_realloc_hook`
: Called at entry to `realloc`, before the actual reallocation. block is a pointer 4 bytes before the address passed to `free` by the application, i.e. it is a pointer to the beginning of the size information maintained before the user buffer. size is the new size requested by the application. (This hook is called *in addition* to the other hooks which will be called by `free` and `malloc` if and when `realloc` calls them.)

The `BLOCK` data type is used by `malloc` and `free` to maintain the heap. The only member which is always guaranteed to be valid is `size` (the additional copy of the size, recorded beyond the buffer's end, is also guaranteed to be valid). The `next` member is valid in all blocks that are part of the free list. This means that `__libc_malloc_hook` can use the `next` member, but `__libc_free_hook` cannot.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

These hooks are specific to DJGPP.

malloc_debug

Syntax

```
#include <stdlib.h>

int malloc_debug(int level);
```

Description

This function sets the level of error diagnosis and reporting during subsequent calls to `malloc`, `free`, `realloc`, and all functions which call them internally. The argument level is interpreted as follows:

Level 0
: No checking; the memory allocation functions behave as they do if `malloc_debug` was never called. Memory in use by the application which was allocated while level 0 was in effect cannot be checked by `malloc_verify` unless it is `free`d first.

Level 1
: Each one of the allocated blocks is recorded in a special structure, where `malloc_verify` can test them for corruption, even if these blocks were not yet `free`d. If errors are detected by `malloc_verify`, it prints diagnostic messages to the standard error stream, with address and size of the offending block and other pertinent information. This level slows down memory allocation to some extent due to additional overhead of calling special functions which record extra debugging info.

Level 2
: Like level 1, but in addition the consistency of the entire heap is verified (by calling `malloc_verify`) on every call to the memory allocation functions. *Warning: this may significantly slow down the application.*

Level 3
: Like level 2, except that the program is aborted whenever a heap corruption is detected. In addition, failed allocations (i.e. when `malloc` returns `NULL` because it cannot satisfy a request) are reported to standard error. Also, if the storage where allocated blocks are recorded is exhausted, a message to that effect is printed.

Level 4
: Like level 3, but calls to `free` with a `NULL` pointer as an argument are also reported.

When `malloc_debug` is first called with a positive argument, it allocates storage for recording blocks in use. To avoid reentrancy problems, this storage is allocated via a direct call to `sbrk`, and its size is fixed. The size used to allocate this storage is by default 400KB, which is enough to record 100 thousand allocated blocks. You can tailor the size to your needs by setting the environment variable `MALLOC_DEBUG` to the maximum number of blocks you want to be able to track. (The value of `MALLOC_DEBUG` should only be as large as the maximum number of allocations which is expected to be in use at any given time, because when a buffer is freed, it is removed from this storage and its cell can be reused for another allocation.) Note that the larger this storage size, the more slow-down will your program experience when the diagnostic level is set to a non-zero value, since the debugging code needs to search the list of recorded blocks in use each time you call `malloc` or `free`.

Return Value

`malloc_debug` returns the previous error diagnostic level. The default level is 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No (see note 1)

Notes:

1. This function is available on many Unix systems.

Example

```
malloc_debug(2);
...
malloc_verify();
```

malloc_verify

Syntax

```
#include <stdlib.h>

int malloc_verify(void);
```

Description

This function attempts to determine if the heap has been corrupted. It scans all the blocks allocated by `malloc` and handed to the application, and also all the free blocks maintained by `malloc` and `free` in the internal free list. Each block is checked for consistency of the hidden bookkeeping information recorded in it by `malloc` and `free`. The blocks on the free list are additionally validated by chasing all the `next` pointers in the linked list and checking them against limits for valid pointers (between 0x1000 and the data segment limit), and the alignment. (Unaligned pointers are probably corrupted, since `malloc` always returns a properly aligned storage.)

What happens when a bad block is found depends on the current malloc diagnostics level: for example, the block can be reported, or the program may be aborted. See malloc_debug, for the details.

Return Value

If the program isn't aborted during the function's run (this depends on the current diagnostics level), `malloc_verify` returns 1 if the heap passes all tests, or zero of some of the tests failed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No (see note 1)

Notes:

1. This function is available on many Unix systems.

Example

```
if (malloc_verify() == 0)
printf ("Heap corruption detected!\n");
```

mallocmap

Syntax

```
#include <stdlib.h>

void mallocmap(void);
```

Description

This function prints a map of the heap storage to standard output. For each block, its address and size are printed, as well as an indication whether it is free or in use. If the slop (a special free block cached for performance reasons) and the small blocks are available, they are printed as well (these two are variants of free blocks). Blocks in use will only be printed if the diagnostic level was set to a non-zero value by a call to `malloc_debug` (See malloc_debug), since otherwise the allocated blocks are not recorded by `malloc`.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No (see note 1)

Notes:

1. This function is available on many Unix systems.

math_errhandling

Syntax

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

Description

`math_errhandling` evaluates to an integer expression describing the floating-point error reporting methods used by the C library.

If `math_errhandling` returns the bit `MATH_ERRNO` set, then errors are reported using `errno` (See errno).

If `math_errhandling` returns the bit `MATH_ERREXCEPT` set, then errors are reported by raising ‘‘exceptions’’.

The library may support both methods of error reporting. Currently DJGPP only supports reporting errors using `errno`.

Return Value

Which floating-point error reporting methods are available.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (math_errhandling & MATH_ERRNO)
perror("myprogram");
```

matherr

Syntax

```
#include <libm/math.h>

enum fdversion _fdlib_version = _SVID_;

int matherr(struct exception *exc);
```

Description

`matherr` is a user-definable handler for errors in math library functions. It is only supported in the alternate math library (link with `-lm`), and will only be called if the global variable `_fdlib_version` is set to either `_SVID_` or `_XOPEN_` (See libm). You also need to mask the Invalid Operation exception in the x87 control word (See _control87) or install a handler for signal `SIGFPE` (See signal), or else some exceptions will generate `SIGFPE` and your program will be terminated before it gets a chance to call `matherr`. DJGPP versions 2.02 and later mask all FP exceptions at startup, so this consideration applies only to programs that unmask FP exceptions at run time.

If the above conditions are met, every math function will call `matherr` when a numerical exception is detected. The default version of `matherr`, supplied with `libm.a`, does nothing and returns zero (the `_SVID_` version will then print an error message to the standard error stream and set `errno`).

This default behavior is inappropriate in some cases. For example, an interactive program which runs in a windowed environment might want the error message to go to a particular window, or pop up a dialog box; a fault-tolerant program might want to fall back to backup procedures so that meaningful results are returned to the application code, etc. In such cases, you should include your own version of `matherr` in your program.

`matherr` is called with a single argument exc which is a pointer to a structure defined on `<libm/math.h>` like this:

```
struct exception {
int type;
char *name;
double arg1, arg2, retval;
};
```

The member `type` is an integer code describing the type of exception that has occured. It can be one of the following:

`DOMAIN`
Argument(s) are outside the valid function domain (e.g., `log(-1)`).

`SING`
Argument(s) would result in a singularity (e.g., `log(0)`).

`OVERFLOW`
The result causes overflow, like in `exp(10000)`.

`UNDERFLOW`
The result causes underflow, like in `exp(-10000)`.

`TLOSS`
The result loses all significant digits, like in `sin(10e100)`.

These codes are defined on `<libm/math.h>`.

The member `name` points to the string that is the name of the function which generated the exception. The members `arg1` and `arg2` are the values of the arguments with which the function was called (`arg2` is undefined if the function only accepts a single argument). The member `retval` is set to the default value that will be returned by the math library function; `matherr` can change it to return a different value.

Return Value

`matherr` should return zero if it couldn't handle the exception, or non-zero if the exception was handled.

If `matherr` returns zero, under `_SVID_` version an error message is printed which includes the name of the function and the exception type, and under `_SVID_` and `_XOPEN_` `errno` is set to an appropriate value. If `matherr` returns non-zero, no error message is printed and `errno` is left unchanged.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <libm/math.h>

int matherr(register struct exception *x)
{
switch (x->type) {
case DOMAIN:
/* change sqrt to return sqrt(-arg1), not NaN */
if (!strcmp(x->name, "sqrt")) {
x->retval = sqrt(-x->arg1);
return 1; /* be silent: no message, don't set errno */
} /* FALL THROUGH */
case SING:
/* all other domain or sing exceptions,
* print message and abort */
```

```
fprintf(stderr, "domain exception in %s\n", x->name);
abort();
break;
}
return 0; /* all other exceptions, execute default procedure */
}
```

mblen

Syntax

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

Description

This function returns the number of characters of string s that make up the next multibyte character. No more than n characters are checked.

If s is `NULL`, the internal shift state is reset.

Return Value

The number of characters that comprise the next multibyte character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int n = mblen(string, INT_MAX);
string += n;
```

mbstowcs

Syntax

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);
```

Description

Converts a multibyte string to a wide character string. The result will be no more than n wide characters.

Return Value

The number of wide characters stored.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int wlen = mbtowcs(wbuf, string, sizeof(wbuf)/sizeof(wchar_t));
```

mbtowc

Syntax

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

Convert the first multibyte sequence in s to a wide character. At most n characters are checked. If pwc is not

`NULL`, the result is stored there. If s is null, the internal shift state is reset.

Return Value

The number of characters used by the multibyte sequence.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
string += mbtowc(&wc, string, strlen(string));
```

_media_type

Syntax

```
#include <dos.h>

int _media_type( const int drive );
```

Description

This function checks if drive number drive (1 == A:, 2 == B:, etc., 0 == default drive) is fixed or removable.

`_media_type` should only be called after you are sure the drive isn't a CD-ROM or a RAM disk, since these might fool you with this call.

Return Value

1 if the drive is a fixed disk, 0 if it is removable. -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{


if( _media_type( 'C' - 'A' + 1 ) )
{
printf("C: is (probably) a hard drive.\n");
}
else
{
printf("C: is (probably) a removable drive.\n");
}

exit(0);
}
```

memalign

Syntax

```
#include <stdlib.h>

void *memalign(size_t size, size_t alignment);
```

Description

This function is like `malloc` (See malloc) except the returned pointer is a multiple of alignment. alignment must be a power of 2.

Return Value

A pointer to a newly allocated block of memory.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *page = memalign(1024, 1024 * 4);
```

memccpy

Syntax

```
#include <string.h>

void * memccpy(void *to, const void *from, int ch, size_t nbytes)
```

Description

This function copies characters from memory area from into to, stopping after the first occurrence of character ch has been copied, or after nbytes characters have been copied, whichever comes first. The buffers should not overlap.

Return Value

A pointer to the character after the copy of ch in to, or a `NULL` pointer if ch was not found in the first nbytes characters of from.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char inpbuf[256], dest[81];

printf("Enter a path: ");
fflush(stdout);
gets(inpbuf);
memset(dest, 0, sizeof(dest));
if (memccpy(dest, inpbuf, '\\', 80))
printf("The first directory in path is %s\n", dest);
else
printf("No explicit directory in path\n");
```

memchr

Syntax

```
#include <string.h>

void *memchr(const void *string, int ch, size_t num);
```

Description

This function searches num bytes starting at string, looking for the first occurence of ch.

Return Value

A pointer to the first match, or `NULL` if it wasn't found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (memchr(path, '/', strlen(path))
do_slash();
```

memcmp

Syntax

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t num);
```

Description

This function compares two regions of memory, at s1 and s2, for num bytes.

Return Value

zero
: s1 == s2

positive
: s1 > s2

negative
: s1 < s2

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

memcpy

Syntax

```
#include <string.h>

void *memcpy(void *dest, const void *src, int num);
```

Description

This function copies num bytes from source to dest. It assumes that the source and destination regions don't overlap; if you need to copy overlapping regions, use `memmove` instead. See memmove.

Return Value

dest

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
memcpy(buffer, temp_buffer, BUF_MAX);
```

memicmp

Syntax

```
#include <string.h>

int memicmp(const void *s1, const void *s2, size_t num);
```

Description

This function compares two regions of memory, at s1 and s2, for num bytes, disregarding case.

Return Value

Zero if they’re the same, nonzero if different, the sign indicates "order".

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (memicmp(arg, "-i", 2) == 0) /* ’-I’ or ’-include’ etc. */
do_include();
```

memmove

Syntax

```
#include <string.h>

void *memmove(void *dest, const void *source, int num);
```

Description

This function copies num bytes from source to dest. The copy is done in such a way that if the two regions overlap, the source is always read before that byte is changed by writing to the destination.

Return Value

dest

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
memmove(buf+1, buf, 99);
memmove(buf, buf+1, 99);
```

memset

Syntax

```
#include <string.h>

void *memset(void *buffer, int ch, size_t num);
```

Description

This function stores num copies of ch, starting at buffer. This is often used to initialize objects to a known value, like zero.

Note that, although ch is declared `int` in the prototype, `memset` only uses its least-significant byte to fill buffer.

Return Value

buffer

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
struct tm t;
memset(&t, 0, sizeof(t));
```

mkdir

Syntax

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

Description

This function creates a subdirectory.

All the bits except `S_IWUSR` in the mode argument are ignored under MS-DOS. If `S_IWUSR` is *not* set in mode, the directory is created with read-only attribute bit set. Note that DOS itself ignores the read-only bit of directories, but some programs do not.

Return Value

Zero if the subdirectory was created, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
mkdir("/usr/tmp", S_IWUSR);
```

mkfifo

Syntax

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

Description

This function is provided only to assist in porting from Unix. It always returns an error condition.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

mknod

Syntax

```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

Description

This function is provided to assist in porting from Unix. If mode specifies a regular file, `mknod` creates a file using path as its name. If mode specifies a character device, and if the device whose name is given by path exists and its device specification as returned by `stat` or `fstat` is equal to dev, `mknod` returns -1 and sets `errno` to `EEXIST`. In all other cases, -1 is returned `errno` is set to `EACCES`.

The argument dev is ignored if mode does not specify a character device.

Return Value

Zero on success, -1 on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

mkstemp

Syntax

```
#include <stdio.h>

int mkstemp(char *template);
```

Description

template is a file specification that ends with six trailing X characters. This function replaces the XXXXXX with a set of characters such that the resulting file name names a nonexisting file. It then creates and opens the file in a way which guarantees that no other process can access this file.

Note that since MS-DOS is limited to eight characters for the file name, and since none of the X's get replaced by a dot, you can only have two additional characters before the X's.

Note also that the path you give will be modified in place.

Return Value

The open file descriptor.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char path[100];
strcpy(path, "/tmp/ccXXXXXX");
int fd = mkstemp(path);
```

mktemp

Syntax

```
#include <stdio.h>

char *mktemp(char *template);
```

Description

template is a file specification that ends with six trailing X characters. This function replaces the XXXXXX with a set of characters such that the resulting file name names a nonexisting file.

Note that since MS-DOS is limited to eight characters for the file name, and since none of the X's get replaced by a dot, you can only have two additional characters before the X's.

Return Value

If a unique name cannot be chosen, NULL is returned. Otherwise the resulting filename is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char template[] = "/tmp/ccXXXXXX";
if (mktemp(template) != NULL)
{
FILE *q = fopen(template, "w");
…
}
```

mktime

Syntax

```
#include <time.h>

time_t mktime(struct tm *tptr);
```

Description

This function converts a time structure into the number of seconds since 00:00:00 GMT 1/1/1970. It also attempts to normalize the fields of tptr. The layout of a `struct tm` is as follows:

```
struct tm {
int tm_sec; /* seconds after the minute [0-60] */
int tm_min; /* minutes after the hour [0-59] */
int tm_hour; /* hours since 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 */
long tm_gmtoff; /* offset from GMT in seconds */
char * tm_zone; /* timezone abbreviation */
};
```

If you don't know whether daylight saving is in effect at the moment specified by the contents of tptr, set the `tm_isdst` member to -1, which will cause `mktime` to compute the DST flag using the data base in the `zoneinfo` subdirectory of your main DJGPP installation. This requires that you set the environment variable `TZ` to a file in that directory which corresponds to your geographical area.

Return Value

The resulting time, or -1 if the time in tptr cannot be described in that format.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

modf

Syntax

```
#include <math.h>

double modf(double x, double *pint);
```

Description

`modf` breaks down x into its integer portion (which it stores in *pint) and the remaining fractional portion, which it returns. Both integer and fractional portions have the same sign as x, except if x is a negative zero, in which case the integer part is a positive zero.

Return Value

The fractional portion. If x is `Inf` or `NaN`, the return value is zero, the integer portion stored in *pint is the same as the value of x, and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

modfl

Syntax

```
#include <math.h>

long double modfl(long double x, long double *pint);
```

Description

`modfl` breaks down x into its integer portion (which it stores in *pint) and the remaining fractional portion, which it returns.

Return Value

The fractional portion.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

moncontrol

Syntax

```
#include <sys/gmon.h>

int moncontrol (int mode);
```

Description

This function allows to control collection of profiling information during the program run. Profiling begins when a program linked with the `-pg` option starts, or when `monstartup` is called (See monstartup). To stop the collection of histogram ticks and function call counts, call `moncontrol` with a zero argument mode; this stops the timer used to sample the program counter (`EIP`) values and disables the code which counts how many times each function compiled with `-pg` was called. To resume collection of profile data, call `moncontrol` with a non-zero argument.

Note that the profiling output is always written to the file `gmon.out` at program exit time, regardless of whether profiling is on or off.

Return Value

`moncontrol` returns the previous state of profiling: zero if it was turned off, non-zero if it was on.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
extern void my_func();
extern void my_func_end();
/* Profile only one function. */
monstartup((unsigned long)my_func, (unsigned long)my_func_end);
...
/* Stop profiling. */
moncontrol(0);
...
/* Resume profiling. */
moncontrol(1);
```

_mono_clear

Syntax

```
#include <sys/mono.h>

void _mono_clear(void);
```

Description

Clears the monochrome monitor.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_mono_printf

Syntax

```
#include <sys/mono.h>

void _mono_printf(const char *fmt, ...);
```

Description

Like See printf, but prints to the monochrome monitor.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_mono_putc

Syntax

```
#include <sys/mono.h>

void _mono_putc(int c);
```

Description

Prints a single character to the monochrome monitor.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

monstartup

Syntax

```
#include <sys/gmon.h>

int monstartup (unsigned long lowpc, unsigned long highpc);
```

Description

This function allows to selectively collect profiling information for a specific range of addresses. The arguments specify the address range that is to be sampled: the lowest address is given by lowpc and the highest is just below highpc. `monstartup` arranges for the profiling data to be gathered and written at program exit time, and then calls the `moncontrol` function (See moncontrol) to start profiling.

The call graph printed by the `gprof` utility will only include functions in this range compiled with the `-pg` option, but `EIP` sampling triggered by the timer tick will measure execution time of all the functions in the specified range.

This function should be called by a program which was not linked with the `-pg` linker switch. If `-pg` *was* used during linking, `monstartup` is called automatically by the startup code with arguments which span the entire range of executable addresses in the program, from the program's entry point to the highest code segment address.

Only the first call to this function has an effect; any further calls will do nothing and return a failure indication. (In particular, in a program linked with `-pg`, this function always fails, since the startup code already called it.) This is because `monstartup` sets up some internal data structures which cannot be resized if a different address range is requested.

Profiling begins on return from this function. You can use `moncontrol` (See moncontrol) to turn profiling off and on.

Return Value

Zero on success, non-zero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

See moncontrol.

movedata

Syntax

```
#include <sys/movedata.h>

void movedata(unsigned source_selector, unsigned source_offset,
unsigned dest_selector, unsigned dest_offset,
size_t length);
```

Description

This function allows the caller to directly transfer information between conventional and linear memory, and among each as well. The selectors passed are *not* segment values like in DOS. They are protected mode selectors that can be obtained by the `_my_ds` and `_go32_info_block.selector_for_linear_memory` (or just `_dos_ds`, which is defined in the include file go32.h) functions (See _my_ds, See _go32_info_block). The offsets are linear offsets. If the selector is for the program's data area, this offset corresponds to the address of a buffer (like `(unsigned)&something`). If the selector is for the conventional memory area, the offset is the physical address of the memory, which can be computed from a traditional segment/offset pair as `segment`*16+`offset`. For example, the color text screen buffer is at offset 0xb8000.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
short blank_row_buf[ScreenCols()];
/* scroll screen */
movedata(_dos_ds, 0xb8000 + ScreenCols()*2,
_dos_ds, 0xb8000,
ScreenCols() * (ScreenRows()-1) * 2);
/* fill last row */
movedata(_my_ds(), (unsigned)blank_row_buf,
_dos_ds, 0xb8000 + ScreenCols()*(ScreenRows()-1)*2,
ScreenCols() * 2);
```

movedatab

Syntax

```
#include <sys/movedata.h>

void _movedatab(unsigned, unsigned, unsigned, unsigned, size_t);
```

Description

Just like See movedata, but all transfers are always 8-bit transfers.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

movedatal

Syntax

```
#include <sys/movedata.h>

void _movedatal(unsigned, unsigned, unsigned, unsigned, size_t);
```

Description

Just like See movedata, but all transfers are always 32-bit transfers, and the count is a count of transfers, not bytes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

movedataw

Syntax

```
#include <sys/movedata.h>

void _movedataw(unsigned, unsigned, unsigned, unsigned, size_t);
```

Description

Just like See movedata, but all transfers are always 16-bit transfers, and the count is a count of transfers, not bytes.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

movetext

Syntax

```
#include <conio.h>

int movetext(int _left, int _top, int _right, int _bottom,
int _destleft, int _desttop);
```

Description

Moves a block of text on the screen.

Return Value

1 on success, zero on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

mprotect

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

Description

This function modifies the access protection of a memory region. Protection occurs in 4Kbyte regions (pages) aligned on 4Kbyte boundaries. All pages in the region will be changed, so addr and len should be multiples of 4096.

The protection prot for each page is specified with the values: PROT_NONE Region can not be touched (if or'ed is ignored). PROT_READ Region can be read (can be or'ed with PROT_WRITE). PROT_WRITE Region can be written (implies read access). This function is only supported on DPMI hosts which provide some V1.0 extensions on V0.9 memory blocks.

Return Value

The function returns 0 if successful and the value -1 if all the pages could not be set.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
mprotect(readonly_buffer,8192,PROT_READ);
mprotect(guard_area,4096,PROT_NONE);
mprotect(NULL,4096,PROT_WRITE); /* Let NULL pointers not generate
* exceptions */
```

_my_cs

Syntax

```
#include <sys/segments.h>

unsigned short _my_cs();
```

Description

Returns the current `CS`. This is useful for setting up interrupt vectors and such.

Return Value

`CS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_my_ds

Syntax

```
#include <sys/segments.h>

unsigned short _my_ds();
```

Description

Returns the current `DS`. This is useful for setting up interrupt vectors and such.

Return Value

`DS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_my_ss

Syntax

```
#include <sys/segments.h>

unsigned short _my_ss();
```

Description

Returns the current `SS`. This is useful for setting up interrupt vectors and such.

Return Value

`SS`

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

nan

Syntax

```
#include <math.h>

double nan(const char *tagp);
```

Description

`nan` returns a quiet NaN with contents indicated by tagp.

Return Value

The quiet NaN.

If quiet NaNs are not supported, zero is returned. DJGPP supports quiet NaNs.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
double d = nan("0x1234");
```

nanf

Syntax

```
#include <math.h>

float nanf(const char *tagp);
```

Description

`nanf` returns a quiet NaN with contents indicated by tagp.

Return Value

The quiet NaN.

If quiet NaNs are not supported, zero is returned. DJGPP supports quiet NaNs.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
float f = nanf("0x1234");
```

nanl

Syntax

```
#include <math.h>

long double nanl(const char *tagp);
```

Description

`nanl` returns a quiet NaN with contents indicated by tagp.

Return Value

The quiet NaN.

If quiet NaNs are not supported, zero is returned. DJGPP supports quiet NaNs.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89

Example

```
long double ld = nanl("0x1234");
```

nice

Syntax

```
#include <unistd.h>

int nice(int _increment);
```

Description

Adjusts the priority of the process. Provided for Unix compatibility only.

Return Value

The new nice value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

normvideo

Syntax

```
#include <conio.h>

void normvideo(void);
```

Description

Resets the text attribute to what it was before the program started.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

nosound

Syntax

```
#include <pc.h>

void nosound(void);
```

Description

Disable the PC speaker.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

ntohl

Syntax

```
#include <netinet/in.h>

unsigned long ntohl(unsigned long val);
```

Description

This function converts from network formatted longs to host formatted longs. For the i386 and higher processors, this means that the bytes are swapped from 1234 order to 4321 order.

Return Value

The host-order value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ip = ntohl(packet.ipaddr);
```

ntohs

Syntax

```
#include <netinet/in.h>

unsigned short ntohs(unsigned short val);
```

Description

This function converts from network formatted shorts to host formatted shorts. For the i386 and higher processors, this means that the bytes are swapped from 12 order to 21 order.

Return Value

The host-order value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
port = ntohs(tcp.port);
```

open

Syntax

```
#include <fcntl.h>
#include <sys/stat.h> /* for mode definitions */

int open(const char *file, int mode /*, int permissions */);
```

Description

This function opens the named file in the given mode, which is any combination of the following bits:

`O_RDONLY`
: The file is opened for reading.

`O_WRONLY`
: The file is opened for writing.

`O_RDWR`
: The file is opened for both reading and writing.

`O_CREAT`
: If the file does not exist, it is created. See creat.

`O_TRUNC`
: If the file does exist, it is truncated to zero bytes.

`O_EXCL`
: If the file exists, and `O_CREAT` is also specified, the `open` call will fail. If the file is a symlink and `O_CREAT` is also specified, then the contents of the symlink are ignored - the `open` call will fail.

`O_APPEND`
: The file pointer is positioned at the end of the file before each write.

`O_TEXT`
: The file is opened in text mode, meaning that Ctrl-M characters are stripped on reading and added on writing as needed. The default mode is specified by the `_fmode` variable See _fmode.

`O_BINARY`
: The file is opened in binary mode.

When called to open the console in binary mode, `open` will disable the generation of `SIGINT` when you press **Ctrl-C** (**Ctrl-Break** will still cause `SIGINT`), because many programs that use binary reads from the console will also want to get the `^C` characters. You can use the `__djgpp_set_ctrl_c` library function (See __djgpp_set_ctrl_c) if you want **Ctrl-C** to generate interrupts while console is read in binary mode.

`O_NOINHERIT`
: Child processes will not inherit this file handle. This is also known as close-on-exec--see See fcntl. This bit is DOS- and Windows-specific; portable programs should use `fcntl` instead.

`O_NOFOLLOW`
: `open` will fail with `errno` set to `ELOOP`, if the last path component in file is symlink.

`O_NOLINK`
: If file is a symlink, `open` will open symlink file itself instead of referred file.

`O_TEMPORARY`
: Delete file when all file descriptors that refer to it are closed.

Note that file should not also be opened with the low-level functions `_creat`, `_creatnew`, `_dos_creat`, `_dos_creatnew`, and `_dos_open`. Otherwise file may not be deleted as expected.

If the file is created by this call, it will be given the read/write permissions specified by permissions, which may be any combination of these values:

`S_IRUSR`
: The file is readable. This is always true for MS-DOS.

`S_IWUSR`
: The file is writable.

Other `S_I*` values may be included, but they will be ignored.

You can specify the share flags (a DOS specific feature) in mode. And you can indicate default values for the share flags in `__djgpp_share_flags`. See __djgpp_share_flags.

You can open directories using `open`, but there is limited support for POSIX file operations on directories. In particular, directories cannot be read using `read` (See read) or written using `write` (See write). The principal reason for allowing `open` to open directories is to support changing directories using `fchdir` (See fchdir). If you wish to read the contents of a directory, use the `opendir` (See opendir) and `readdir` (See readdir) functions instead. File descriptors for directories are not inherited by child programs.

Return Value

If successful, the file descriptor is returned. On error, a negative number is returned and `errno` is set to indicate the error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
int q = open("/tmp/foo.dat", O_RDONLY|O_BINARY);
```

_open

Syntax

```
#include <io.h>

int _open(const char *path, int attrib);
```

Description

This is a direct connection to the MS-DOS open function call, int 0x21, %ah = 0x3d, on versions of DOS earlier than 7.0. On DOS version 7.0 or later `_open` calls function int 0x21, %ax = 0x6c00. When long file names are supported, `_open` calls function 0x716c of Int 0x21.

On FAT32 file systems file sizes up to 2^32-2 are supported. Note that WINDOWS 98 has a bug which only lets you create these big files if LFN is enabled. In plain DOS mode it plainly works.

The file is set to binary mode.

The attrib parameter is a combination of one or more bits from the following:

`O_RDONLY`
: open for read only

`O_WRONLY`
: open for write only

`O_RDWR`
: open for read and write

`O_NOINHERIT`
: file handle is not inherited by child processes

`SH_COMPAT`
: open in compatibility mode

`SH_DENYRW`
: deny requests by other processes to open the file for eaither reading or writing

`SH_DENYWR`
: deny requests to open the file for writing

`SH_DENYRD`
: deny requests to open the file for reading

`SH_DENYNO`
: deny-none mode: allow other processes to open the file if their open mode doesn't conflict with the open mode of this process

This function can be hooked by File System Extensions (See File System Extensions). If you don't want this, you should use `_dos_open` (See _dos_open) (but note that the latter doesn't support long file names).

Return Value

The new file descriptor, else -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

opendir

Syntax

```
#include <dirent.h>

extern int __opendir_flags;

DIR *opendir(char *name);
```

Description

This function "opens" a directory so that you can read the list of file names in it. The pointer returned must be passed to `closedir` when you are done with it. See readdir.

The global variable `__opendir_flags` can be set to include the following values to control the operation of `opendir`:

`__OPENDIR_PRESERVE_CASE`
: Do not change the case of files to lower case. Just in case Micros*ft decides to support case-sensitive file systems some day.

  You can also use this flag if you want the names of files like `README` and `FAQ` from Unix distributions to be returned in upper-case on Windows 9X filesystems. See _preserve_fncase, for other ways of achieving this and for more detailed description of the automatic letter-case conversion by DJGPP library functions.

`__OPENDIR_NO_HIDDEN`
: Do not include hidden files and directories in the search. By default, all files and directories are included.

`__OPENDIR_FIND_HIDDEN`
: Provided for back-compatibility with previous DJGPP versions, where hidden files and directories were by default skipped. In versions 2.02 and later, this flag has no effect.

`__OPENDIR_FIND_LABEL`
: Include volume labels in the search. By default, these are skipped.

`__OPENDIR_NO_D_TYPE`
: Do not compute the `d_type` member of `struct dirent`. If this flag is set, all files will get `DT_UNKNOWN` in the `d_type` member. By default, this flag is reset. See readdir.

You can simply put `int __opendir_flags = ...;` in your code. The default is to let it get set to zero as an uninitialized variable.

Return Value

The open directory structure, or `NULL` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. The `__opendir_flags` variable is DJGPP-specific.

Example

```
DIR *d = opendir(".");
closedir(d);
```

outb

Syntax

```
#include <pc.h>

void outb(unsigned short _port, unsigned char _data);
```

Description

Calls See outportb. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outp

Syntax

```
#include <pc.h>

void outp(unsigned short _port, unsigned char _data);
```

Description

Calls See outportb. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportb

Syntax

```
#include <pc.h>

void outportb(unsigned short _port, unsigned char _data);
```

Description

Write a single byte to an 8-bit port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportl

Syntax

```
#include <pc.h>

void outportl(unsigned short _port, unsigned long _data);
```

Description

Write a single long to an 32-bit port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportsb

Syntax

```
#include <pc.h>

void outportsb(unsigned short _port,
const unsigned char *_buf, unsigned _len);
```

Description

Writes the _len bytes in _buf to the 8-bit _port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportsl

Syntax

```
#include <pc.h>

void outportsl(unsigned short _port,
const unsigned long *_buf, unsigned _len);
```

Description

Writes the _len longs in _buf to the 32-bit _port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportsw

Syntax

```
#include <pc.h>

void outportsw(unsigned short _port,
const unsigned short *_buf, unsigned _len);
```

Description

Writes the _len shorts in _buf to the 16-bit _port.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outportw

Syntax

```
#include <pc.h>

void outportw(unsigned short _port, unsigned short _data);
```

Description

Write a single short to an 16-bit port.

This function is provided as an inline assembler macro, and will be optimized down to a single opcode when you optimize your program.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

outpw

Syntax

```
#include <pc.h>

void outpw(unsigned short _port, unsigned short _data);
```

Description

Calls See outportw. Provided only for compatibility.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

pathconf

Syntax

```
#include <unistd.h>

long pathconf(const char *filename, int name);
```

Description

This function returns various system-dependent configuration values. The name is one of the following:

`_PC_LINK_MAX`
: The maximum number of directory entries that can refer to a single real file. Always 1 in DJGPP.

`_PC_MAX_CANON`
: The maximum number of bytes in an editable input line. In DJGPP, this is 126 (DOS restriction).

`_PC_MAX_INPUT`
: The maximum number of bytes in a non-editable input line. Also 126 in DJGPP.

`_PC_NAME_MAX`
: The maximum length of an individual file name. If the filesystem where filename resides supports long file names, the result is whatever `_get_volume_info` returns (usually 255); otherwise 12 will be returned. See _use_lfn.

`_PC_PATH_MAX`
: The maximum length of a complete path name. If the filesystem where filename resides supports long file names, the result is whatever `_get_volume_info` returns (usually 260); otherwise 80 will be returned. See _use_lfn.

`_PC_PIPE_BUF`
: The size of a pipe's internal buffer. In DJGPP, this returns 512.

`_PC_CHOWN_RESTRICTED`
: If non-zero, only priviledged user can change the ownership of files by calling `chown`, otherwise anyone may give away files. The DJGPP version always returns zero, since MS-DOS files can be freely given away.

`_PC_NO_TRUNC`
: If zero is returned, filenames longer than what `pathconf (filename, _PC_NAME_MAX)` returns are truncated, otherwise an error occurs if you use longer names. In DJGPP, this returns 0, since DOS always silently truncates long names.

`_PC_VDISABLE`
: A character to use to disable tty special characters. DJGPP currently doesn't support special characters, so this returns -1.

Return Value

The selected configuration value is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
char *buf = malloc(pathconf("c:/", _PC_MAX_PATH)+1);
```

pause

Syntax

```
#include <unistd.h>

int pause(void);
```

Description

This function just calls `__dpmi_yield()` (See __dpmi_yield) to give up a slice of the CPU.

Return Value

Zero.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

pclose
Syntax
```
#include <stdio.h>

int pclose(FILE *pipe);
```

Description
This function closes a pipe opened with `popen` (See popen). Note that since MS-DOS is not multitasking, this function will actually run the program specified in `popen` if the pipe was opened for writing.

Return Value
Zero on success, nonzero on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example
```
FILE *f = popen("sort", "w");
write_to_pipe(f);
pclose(f);
```

perror
Syntax
```
#include <stdio.h>

void perror(const char *string);
```

Description
This function formats an error message and prints it to `stderr`. The message is the string, a colon and a blank, and a message suitable for the error condition indicated by `errno`. If string is a null pointer or points to a null string, the colon and blank are not printed.

Return Value
None.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example
```
int x = open("foo", O_RDONLY);
if (x < 0)
{

perror("foo");
exit(1);
}
```

pipe
Syntax
```
#include <unistd.h>
```

```
int pipe(int fildes[2]);
```

Description

This function creates a pipe and places a file descriptor for the read end of the pipe in `fildes[0]`, and another for the write end in `fildes[1]`. Data written to `fildes[1]` will be read from `fildes[0]` on a first-in first-out (FIFO) basis.

Note this pipe implementation won't help port instances of `fork`/`exec` or any other methods that require support for multitasking.

Return Value

Zero for success, otherwise -1 is returned and `errno` is set to indicate the error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

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

/* Pipe the output of program to the input of another. */

int main()
{

int pipe_fds[2];
int stdin_save, stdout_save;
if (pipe(pipe_fds) < 0)
return -1;

/* Duplicate stdin and stdout so we can restore them later. */
stdin_save = dup(STDIN_FILENO);
stdout_save = dup(STDOUT_FILENO);

/* Make the write end of the pipe stdout. */
dup2(pipe_fds[1], STDOUT_FILENO);

/* Run the program. Its output will be written to the pipe. */
spawnl(P_WAIT, "/dev/env/DJDIR/bin/ls.exe", "ls.exe", NULL);

/* Close the write end of the pipe. */
close(pipe_fds[1]);

/* Restore stdout. */
dup2(stdout_save, STDOUT_FILENO);

/* Make the read end of the pipe stdin. */
dup2(pipe_fds[0], STDIN_FILENO);

/* Run another program. Its input will come from the output of the
first program. */
spawnl(P_WAIT, "/dev/env/DJDIR/bin/less.exe", "less.exe", "-E", NULL);

/* Close the read end of the pipe. */
close(pipe_fds[0]);

/* Restore stdin. */
dup2(stdin_save, STDIN_FILENO);
return 0;
}
```

popen

Syntax

```
#include <stdio.h>

FILE *popen(const char *cmd, const char *mode);
```

Description

This function executes the command or program specified by `cmd` and attaches either its input stream or its output stream to the returned file. While the file is open, the calling program can write to the program (if the program was open for writing) or read the program's output (if the program was opened for reading). When the program is done, or if you have no more input for it, pass the file pointer to `pclose` (See pclose), which terminates the program.

Since MS-DOS does not support multitasking, this function actually runs the entire program when the program is opened for reading, and stores the output in a temporary file. `pclose` then removes that file. Similarly, when you open a program for writing, a temp file holds the data and `pclose` runs the entire program.

The mode is the same as for `fopen` (See fopen), except that you are not allowed to open a pipe for both reading and writing. A pipe can be open either for reading or for writing.

Return Value

An open file which can be used to read the program's output or write to the program's input.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
FILE *p = popen("dir", "r");
read_program(p);
pclose(p);
```

pow

Syntax

```
#include <math.h>

double pow(double x, double y);
```

Description

This function computes x^y, x raised to the power y.

Return Value

x raised to the power y. If the result overflows a `double` or underflows, `errno` is set to `ERANGE`. If y is `NaN`, the return value is `NaN` and `errno` is set to `EDOM`. If x and y are both 0, the return value is 1, but `errno` is set to `EDOM`. If y is a positive or a negative Infinity, the following results are returned, depending on the value of x:

x negative
: the return value is `NaN` and `errno` is set to `EDOM`.

absolute value of x less than 1 and y is `+Inf`
absolute value of x greater than 1 and y is `-Inf`
: the return value is zero.

absolute value of x less than 1 and y is `-Inf`
absolute value of x greater than 1 and y is `+Inf`
: the return value is `+Inf`.

absolute value of x is 1
: the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

pow10

Syntax

```
#include <math.h>

double pow10(double x);
```

Description

This function computes 10 to the power of x, 10^x.

Return Value

10 to the x power. If the value of x is finite, but so large in magnitude that 10^x cannot be accurately represented by a `double`, the return value is the nearest representable ! `double` (possibly, an `Inf`), and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or zero, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

pow2

Syntax

```
#include <math.h>

double pow2(double x);
```

Description

This function computes 2 to the power of x, 2^x.

Return Value

2 to the x power. If the value of x is finite, but so large in magnitude that 2^x cannot be accurately represented by a `double`, the return value is the nearest representable `double` (possibly, an `Inf`), and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is either `+Inf` or zero, respectively, and `errno` is not changed. If x is a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

powi

Syntax

```
#include <math.h>

double powi(double x, int iy);
```

Description

This function computes x^iy, where iy is an integer number. It does so by an optimized sequence of squarings and multiplications. For integer values of exponent, it is always faster to call `powi` than to call `pow` with the same arguments, even if iy has a very large value. For small values of iy, `powi` is *much* faster than `pow`.

Return Value

x raised to the iy power. If x and iy are both zero, the return value is 1. If x is equal to zero, and iy is negative, the return value is `Inf`. This function never sets `errno`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_preserve_fncase

Syntax

```
#include <fcntl.h>

char _preserve_fncase (void);
```

Description

This function returns a non-zero value if letter-case in filenames should be preserved. It is used by library functions that get filenames from the operating system (like `readdir`, `_fixpath` and others). The usual behavior of these functions (when `_preserve_fncase` returns zero) is to down-case 8+3 DOS-style filenames, but leave alone the letter-case in long filenames when these are supported (See _use_lfn). This can be changed by either setting `_CRT0_FLAG_PRESERVE_FILENAME_CASE` bit in the `_crt0_startup_flags` variable (See _crt0_startup_flags), or by setting the `FNCASE` environment variable to **`Y`** at run time. You might need such a setup e.g. on Windows 95 if you want to see files with names like `README` and `FAQ` listed in upper-case (for this to work, you will have to manually rename all the other files with 8+3 DOS-style names to lower-case names). When the case in filenames is preserved, all filenames will be returned in upper case on MSDOS (and other systems that don't support long filenames), or if the environment variable `LFN` is set to **`N`** on systems that support LFN. That is because this is how filenames are stored in the DOS directory entries.

Return value

Zero when 8+3 filenames should be converted to lower-case, non-zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

PRI

Syntax

```
#include <inttypes.h>
```

Description

The `PRI` family of macros allows integers to be displayed in a portable manner using the `printf` family of functions (See printf). They include a conversion qualifier, to specify the width of the type (e.g.: `l` for `long`), and the conversion type specifier (e.g.: `d` for decimal display of integers).

The `PRI` family of macros should be used with the types defined in the header `<stdint.h>`. For example: `int8_t`, `uint_fast32_t`, `uintptr_t`, `intmax_t`.

Below N can be 8, 16, 32 or 64. The `PRI` macros are:

`PRIdN`
`PRIiN`
: The `d` and `i` type conversion specifiers for a type `intN_t` of N bits.

`PRIdLEASTN`
`PRIiLEASTN`
: The `d` and `i` type conversion specifiers for a type `int_leastN_t` of N bits.

`PRIdFASTN`
`PRIiFASTN`
: The `d` and `i` type conversion specifiers for a type `int_fastN_t` of N bits.

`PRIdMAX`
`PRIiMAX`
: The `d` and `i` type conversion specifiers for a type `intmax_t`.

`PRIdPTR`
`PRIiPTR`
: The `d` and `i` type conversion specifier for a type `intptr_t`.

`PRIoN`
`PRIuN`
`PRIxN`
`PRIXN`

The o, u, x and X type conversion specifiers for a type `uintN_t` of N bits.

`PRIoLEASTN`
`PRIuLEASTN`
`PRIxLEASTN`
`PRIXLEASTN`
The o, u, x and X type conversion specifiers for a type `uint_LEASTN_t` of N bits.

`PRIoFASTN`
`PRIuFASTN`
`PRIxFASTN`
`PRIXFASTN`
The o, u, x and X type conversion specifiers for a type `uint_FASTN_t` of N bits.

`PRIoMAX`
`PRIuMAX`
`PRIxMAX`
`PRIXMAX`
The o, u, x and X type conversion specifiers for a type `uintmax_t`.

`PRIoPTR`
`PRIuPTR`
`PRIxPTR`
`PRIXPTR`
The o, u, x and X type conversion specifiers for a type `uintptr_t`.

Return Value

Not applicable.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
uintmax_t foo = 500;

printf("foo is %" PRIuMAX " in decimal and 0x%" PRIxMAX " in hex.\n",
foo, foo);
```

printf

Syntax

```
#include <stdio.h>

int printf(const char *format, ...);
```

Description

Sends formatted output from the arguments (…) to `stdout`.

The format string contains regular characters to print, as well as conversion specifiers, which begin with a percent symbol. Each conversion speficier contains the following fields:

- an optional flag, which may alter the conversion:

  `-`
  left-justify the field.

  `+`
  Force a + sign on positive numbers.

  `space`
  To leave a blank space where a plus or minus sign would have been.

  `#`
  Alternate conversion - prefix octal numbers with 0, hexadecimal numbers with 0x or 0X, or force a

trailing decimal point if a floating point conversion would have omitted it.

0
: To pad numbers with leading zeros.

- A field width specifier, which specifies the minimum width of the field. This may also be an asterisk (`*`), which means that the actual width will be obtained from the next argument. If the argument is negative, it supplies a `-` flag and a positive width.
- An optional decimal point and a precision. This may also be an asterisk, but a negative argument for it indicates a precision of zero. The precision specifies the minimum number of digits to print for an integer, the number of fraction digits for a floating point number (max for `g` or `G`, actual for others), or the maximum number of characters for a string.
- An optional conversion qualifier, which may be:

`hh`
: to specify `char`;

`h`
: to specify `short` integers;

`j`
: to specify `intmax_t` or `uintmax_t` integers;

`l`
: to specify doubles or `long` integers;

`ll`
: (two lower-case ell letters) to specify `long long` integers; to specify `long` doubles, although this is non-standard;

`L`
: to specify `long` doubles;

`t`
: to specify `ptrdiff_t`;

`z`
: to specify `size_t`.

- The conversion type specifier:

`c`
: A single character.

`d`
: A signed integer.

`D`
: A signed long integer. This is non-standard and obsolete. Please use `ld` instead.

`e`
`E`
: A floating point number (float or double). For long double, use `"Le"` or `"LE"`. The exponent case matches the specifier case. The representation always has an exponent.

`f`
: A floating point number (float or double). For long double, use `"Lf"`. The representation never has an exponent.

`g`
`G`
: A floating point number (float or double). For long double, use `"Lg"` or `"LG"`. The exponent case matches the specifier case. The representation has an exponent if it needs one.

`i`
: A signed integer.

`n`
: The next argument is a pointer to an integer, and the number of characters generated so far is stored in that integer.

`o`
: A unsigned integer, printed in base 8 instead of base 10.

`O`
: A unsigned long integer, printed in base 8 instead of base 10. This is non-standard and obsolete. Please use `lo` instead.

`p`
: A pointer. This is printed with an `x` specifier.

`s`
: A `NULL`-terminated string.

`u`
: An unsigned integer.

`U`
: An unsigned long integer. This is non-standard and obsolete. Please use `lu` instead.

`x`
`X`
: An unsigned integer, printed in base 16 instead of base 10. The case of the letters used matches the specifier case.

`%`
: A single percent symbol is printed.

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1) (see note 2) 1003.2-1992; 1003.1-2001

Notes:

1. The `hh`, `j`, `t` and `z` conversion specifiers first appeared in the ANSI C99 standard.
2. The `D`, `O` and `U` conversion types are non-standard. gcc may generate warnings, if you use them.

Example

```
printf("%-3d %10.2f%% Percent of %s\n", index, per[index], name[index]);
```

psignal

Syntax

```
#include <signal.h>

extern char *sys_siglist[];
void psignal (int sig, const char *msg);
```

Description

This function produces a message on the standard error stream describing the signal given by its number in sig. It prints the string pointed to by msg, then the name of the signal, and a newline.

The names of signals can be retrieved using the array `sys_siglist`, with the signal number serving as an index into this array.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <signal.h>

void sig_catcher (int sig)
{
psignal (progname, sig);
return;
}
```

_put_path

Syntax

```
#include <libc/dosio.h>

int _put_path(const char *path);
int _put_path2(const char *path, int offset);
```

Description

These functions are used internally by all low-level library functions that need to pass file names to DOS. `_put_path` copies its argument path to the transfer buffer (See _go32_info_block) starting at the beginning of the transfer buffer; `_put_path2` does the same except that it puts the file name starting at offset bytes from the beginning of the transfer buffer.

These functions are meant to be called by low-level library functions, not by applications. You should only call them if you know what you are doing. In particular, if you call any library function between a call to `_put_path` or `_put_path2` and the call to a DOS function that uses the file name, the file name in the transfer buffer could be wiped out, corrupted or otherwise changed. You *have* been warned!

Some constructs in file names are transformed while copying them, to allow transparent support for nifty features. Here's the list of these transformations:

- Multiple forward slashes are collapsed into a single slash. Unix treats multiple slashes as a single slash, so some ported programs pass names like `c:/foo//bar` to library functions. DOS functions choke on such file names, so collapsing the slashes prevents these names from failing.

- Trailing slashes are removed, except for root directories. Various DOS calls cannot cope with file names like `c:/foo/`; this feature solves this problem.

- Translation of Unix device names. Unix `/dev/null` is mapped to DOS-standard `NUL`, and Unix `/dev/tty` to DOS-standard `CON`. This provides for transparent support of these special devices, e.g. in Unix shell scripts.

- Translation of DOS device names. Any file name which begins with `/dev/` or `x:/dev/` (where x: is any valid DOS drive letter) has the `/dev/` or `x:/dev/` prefix removed (if the `/dev/` directory does not exist), and the rest is passed to DOS. This is because some DOS functions don't recognize device names unless they are devoid of the drive and directory specifications, and programs could add a drive and a directory if they convert a name like `/dev/con` to a fully-qualified path name. Because of the different behavior when the `/dev/` directory exists, you should only add the prefix `/dev/` to your DOS device names if necessary and be sure that the `/dev/` does not exist. Due to the additional overhead of checking if `/dev/` exists, functions working with DOS device names with the prefix will be slower.

- `/dev/x/` is translated into `x:/`. This allows to use Unix-style absolute path names that begin with a slash, instead of DOS-style names with a drive letter. Some Unix programs and shell scripts fail for file names that include colons, which are part of the drive letter specification; this feature allows to work around such problems by using e.g. `/dev/c/` where `c:/` would fail.

- `/dev/env/foo/` is replaced by the value of the environmentvariable foo.

  (In other words, you can think of environment variables as if they were sub-directories of a fictitious directory `/dev/env`.)

  This allows to use environment variable names inside path names compiled into programs, and have them transparently expanded at run time. For example, `/dev/env/DJDIR/include` will expand to the exact path name of the DJGPP include directory, no matter where DJGPP is installed on the machine where the program runs. (The value of `DJDIR` is computed by the DJGPP startup code and pushed into the environment of every DJGPP program before `main` is called.)

Note that environment variables are case-sensitive, so `/dev/env/foo` and `/dev/env/FOO` are **not** the same. DOS shells usually upcase the name of the environment variable if you set it with the built-in command `SET`, so if you type e.g. `SET foo=bar`, the shell defines a variable named `FOO`.

If the environment variable is undefined, it will expand into an empty string. The expansion is done recursively, so environment variables may reference other environment variables using the same `/dev/env/` notation. For example, if the variable `HOME` is set to `/dev/env/DJDIR/home`, and DJGPP is installed in `c:/software/djgpp`, then `/dev/env/HOME/sources` will expand to `c:/software/djgpp/home/sources`.

It is possible to supply a default value, to be used if the variable is not defined, or has an empty value. To this end, put the default value after the name of the variable and delimit it by `~`, like in `/dev/env/DJDIR~c:/djgpp~/include`.

If you need to include a literal character `~` in either the environment variable name or in the default value that replaces it, use two `~`s in a row. For example, `/dev/env/FOO~~` will expand to the value of the variable `FOO~`. Likewise, `/dev/env/FOO~~BAR~foo~~baz~` will expand to the value of the variable `FOO~BAR` if it is defined and nonempty, and to `foo~baz` otherwise. Leading `~` in the default value isn't supported (it is interpreted as part of the preceding variable name).

The default value may also reference (other) environment variables, but nested default values can get tricky. For example, `/dev/env/foo~/dev/env/bar~` will work, but `/dev/env/foo~/dev/env/bar~baz~~` will **not**. To use nested default values, you need to double the quoting of the `~` characters, like in `/dev/env/foo~/dev/env/bar~~baz~~~`.

Return Value

Both functions return the offset into the transfer buffer of the terminating null character that ends the file name.

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

These functions are meant to be called by low-level library functions, not by applications. You should only call them if you know what you are doing. In particular, if you call any library function between a call to `_put_path` or `_put_path2` and the call to a DOS function that uses the file name, the file name in the transfer buffer could be wiped out, corrupted and otherwise changed. You *have* been warned!

```
__dpmi_regs r;

_put_path("/dev/c/djgpp/bin/");
r.x.ax = 0x4300; /* get file attributes */
r.x.ds = __tb >> 4;
r.x.dx = __tb & 0x0f;
__dpmi_int(0x21, &r);
```

putc

Syntax

```
#include <stdio.h>

int putc(int c, FILE *file);
```

Description

This function writes one character to the given file.

Return Value

The character written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
while ((c=getc(stdin)) != EOF)
putc(c, stdout);
```

putch

Syntax

```
#include <conio.h>

int putch(int _c);
```

Description

Put the character _c on the screen at the current cursor position. The special characters return, linefeed, bell, and backspace are handled properly, as is line wrap and scrolling. The cursor position is updated.

Return Value

The character is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

putchar

Syntax

```
#include <stdio.h>

int putchar(int c);
```

Description

This is the same as `fputc(c, stdout)`. See fputc.

Return Value

The character written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
while ((c = getchar()) != EOF)
putchar(c);
```

putenv

Syntax

```
#include <stdlib.h>

int putenv(char *env);
```

Description

This function adds an entry to the program's environment. The string passed must be of the form `NAME=VALUE`. Any existing value for the environment variable is gone.

`putenv` will copy the string passed to it, and will automatically free any existing string already in the environment. Keep this in mind if you alter the environment yourself. The string you pass is still your responsibility to free. Note that most implementations will not let you free the string you pass, resulting in memory leaks.

Return Value

Zero on success, nonzero on failure; `errno` will be set to the relevant error code: currently only `ENOMEM` (insufficient memory) is possible.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. This function is new to the Posix 1003.1-200x draft

Example

```
putenv("SHELL=ksh.exe");
```

puts

Syntax

```
#include <stdio.h>

int puts(const char *string);
```

Description

This function writes string to `stdout`, and then writes a newline character.

Return Value

Nonnegative for success, or `EOF` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
puts("Hello, there");
```

puttext

Syntax

```
#include <conio.h>

int puttext(int _left, int _top, int _right, int _bottom,
void *_source);
```

Description

The opposite of See gettext.

Return Value

1 on success, zero on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

putw

Syntax

```
#include <stdio.h>

int putw(int x, FILE *file);
```

Description

Writes a single 32-bit binary word x in native format to file. This function is provided for compatibility with other 32-bit environments, so it writes a 32-bit `int`, not a 16-bit `short`, like some 16-bit DOS compilers do.

Return Value

The value written, or `EOF` for end-of-file or error. Since `EOF` is a valid integer, you should use `feof` or `ferror` to detect this situation.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
putw(12, stdout);
```

pwrite

Syntax

```
#include <unistd.h>

int pwrite(int file, const void *buffer, size_t count, off_t offset);
```

Description

This function writes count bytes from buffer to file at position offset. It returns the number of bytes actually written. It will return zero or a number less than count if the disk is full, and may return less than count even under valid conditions.

Note that if file is a text file, `pwrite` may write more bytes than it reports.

If count is zero, the function does nothing and returns zero. Use `_write` if you want to actually ask DOS to write zero bytes.

The precise behavior of `pwrite` when the target filesystem is full are somewhat troublesome, because DOS doesn't fail the underlying system call. If your application needs to rely on `errno` being set to `ENOSPC` in such cases, you need to invoke `pwrite` as shown in an example for `write` (See write). In a nutshell, the trick is to call `pwrite` one more time after it returns a value smaller than the count parameter; then it will *always* set `errno` if the disk is full.

Return Value

The number of bytes written, zero at EOF, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.1-2001; not

1003.2-1992

Example

```
const char buf[] = "abc";
const size_t bufsize = strlen(buf);

/* Write out buf, then overwrite ’b’ with ’d’. NB: We should check
* for errors. */
lseek(fd, 0L, SEEK_SET);
write(fd, buf, bufsize);
pwrite(fd, "d", 1, 1);
```

qsort

Syntax

```
#include <stdlib.h>

void qsort(void *base, size_t numelem, size_t size,
int (*cmp)(const void *e1, const void *e2));
```

Description

This function sorts the given array in place. base is the address of the first of numelem array entries, each of size size bytes. `qsort` uses the supplied function cmp to determine the sort order for any two elements by passing the address of the two elements and using the function’s return address.

The return address of the function indicates the sort order:

Negative
: Element e1 should come before element e2 in the resulting array.

Positive
: Element e1 should come after element e2 in the resulting array.

Zero
: It doesn’t matter which element comes first in the resulting array.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
typedef struct {
int size;
int sequence;
} Item;

int qsort_helper_by_size(const void *e1, const void *e2)
{

return ((const Item *)e2)->size - ((const Item *)e1)->size;
}


Item list[100];

qsort(list, 100, sizeof(Item), qsort_helper_by_size);

int qsort_stringlist(const void *e1, const void *e2)
{

return strcmp(*(char **)e1, *(char **)e2);
```

```
}

char *slist[10];

/* alphabetical order */
qsort(slist, 10, sizeof(char *), qsort_stringlist);
```

raise

Syntax

```
#include <signal.h>

int raise(int sig);
```

Description

This function raises the given signal sig. See signal, the list of possible signals.

Return Value

0 on success, -1 for illegal value of sig.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

rand

Syntax

```
#include <stdlib.h>

int rand(void);
```

Description

Returns a pseudo-random number between zero and `RAND_MAX` (defined on `stdlib.h`).

By default, this function always generates the same sequence of numbers each time you run the program. (This is usually desirable when debugging, or when comparing two different runs.) If you need to produce a different sequence on every run, you must seed `rand` by calling `srand` (See srand) before the first call to `rand`, and make sure to use a different argument to `srand` each time. The usual technique is to get the argument to `srand` from a call to the `time` library function (See time), whose return value changes every second.

To get a random number in the range 0..N, use `rand()%(N+1)`. Note that the low bits of the `rand`'s return value are not very random, so `rand()%N` for small values of N could be not enough random. The alternative, but non-ANSI, function `random` is better if N is small. See random.

Return Value

The number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
/* random pause */
srand(time(0));
for (i=rand(); i; i--);
```

rand48

Syntax

```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short state[3]);
unsigned long lrand48(void);
unsigned long nrand48(unsigned short state[3]);
long mrand48(void);
long jrand48(unsigned short state[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short state_seed[3]);
void lcong48(unsigned short param[7]);
```

Description

This is the family of `*rand48` functions. The basis for these functions is the linear congruential formula X[n+1] = (a*X[n] + c) mod 2^48, n >= 0. a = 0x5deece66d and c = 0xb at start and after a call to either `srand48` or `seed48`. A call to `lcong48` changes a and c (and the internal state).

`drand48` and `erand48` return `double`s uniformly distributed in the interval [0.0, 1.0).

`lrand48` and `nrand48` return `unsigned long`s uniformly distributed in the interval [0, 2^31).

`mrand48` and `jrand48` return `long`s uniformly distributed in the interval [-2^31, 2^31).

`erand48`, `jrand48` and `nrand48` requires the state of the random generator to be passed.

`drand48`, `lrand48` and `mrand48` uses an internal state (common with all three functions) which should be initialized with a call to one of the functions `srand48`, `seed48` or `lcong48`.

`srand48` sets the high order 32 bits to the argument seed. The low order 16 bits are set to the arbitrary value 0x330e.

`seed48` sets the internal state according to the argument state_seed (`state_seed[0]` is least significant). The previous state of the random generator is saved in an internal (static) buffer, to which a pointer is returned.

`lcong48` sets the internal state to `param[0-2]`, a to `param[3-5]` (`param[0]` and `param[3]` are least significant) and c to `param[6]`.

Return Value

A random number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

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

int main(void)
{


srand48(time(NULL));
printf("%.12f is a random number in [0.0, 1.0).\n", drand48());

exit(0);
}
```

random

Syntax

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

```
long random(void);
```

Description
Returns a random number in the range 0..MAXINT.

Return Value
0 .. MAXINT

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
/* Produce a random integer between 0 and 199. */
int random_number = random () % 200;
```

rawclock
Syntax
```
#include <time.h>

unsigned long rawclock(void);
```

Description
Returns the number of clock tics (18.2 per second) since midnight.

Return Value
The number of tics.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
/* wait 1/4 second */
int i = rawclock()+5;
while (rawclock()<i);
```

_rdtsc
Syntax
```
#include <time.h>

unsigned long long _rdtsc(void);
```

Description
This function invokes the hardware instruction `rdtsc` which is only supported on some processors. It is incremented once per clock cycle on the main processor. It is a high precision timer which is useful for timing code for optimization. You should not use this function in distributed programs without protecting for processors which do not support the instruction. For a general purpose high precision timer see `uclock` (See uclock).

Return Value
The number of processor cycles.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

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

```
#include <time.h>
#include <signal.h>
#include <setjmp.h>
#include <sys/exceptn.h>

/* Catch rdtsc exception and always return 0LL */
void catch_rdtsc(int val)
{

short *eip = (short *)__djgpp_exception_state->__eip;
if(*eip == 0x310f) {
__djgpp_exception_state->__eip += 2;
__djgpp_exception_state->__edx = 0;
longjmp(__djgpp_exception_state, 0);
}
return;
}


int main(void)
{

unsigned long long t;
signal(SIGILL, catch_rdtsc);
t = _rdtsc();
printf("Timer value: %llu\n",t);
return 0;
}
```

read

Syntax

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t length);
```

Description

This function reads at most length bytes from file fd into buffer. Note that in some cases, such as end-of-file conditions and text files, it may read less than the requested number of bytes. At end-of-file, `read` will read exactly zero bytes.

Directories cannot be read using `read` --- use `readdir` instead.

Return Value

The number of bytes read, zero meaning end-of-file, or -1 for an error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
char buf[10];
int r = read(0, buf, 10);
```

_read

Syntax

```
#include <io.h>

ssize_t _read(int fildes, void *buf, size_t nbyte);
```

Description

This is a direct connection to the MS-DOS read function call, int 0x21, %ah = 0x3f. No conversion is done on the data; it is read as raw binary data. This function can be hooked by the See File System Extensions. If you don't want this, you should use See _dos_read.

Return Value

The number of bytes read.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

read_child

Syntax

```
#include <debug/dbgcom.h>

void read_child (unsigned child_addr, void *buf, unsigned len);
```

Description

This function reads the memory of the debugged process starting at address child_addr for len bytes, and copies the data read to the buffer pointed to by buf. It is used primarily to save the original instruction at the point where a breakpoint instruction is inserted (to trigger a trap when the debuggee's code gets to that point). See write_child.

Return Value

The function return zero if it has successfully transferred the data, non-zero otherwise (e.g., if the address in child_addr is outside the limits of the debuggee's code segment.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

read_sel_addr

Syntax

```
#include <debug/dbgcom.h>

void read_sel_addr (unsigned offset, void *buf, unsigned len,
unsigned sel);
```

Description

This function reads the memory starting at offset offset in selector sel for len bytes, and copies the data read to the buffer pointed to by buf. See write_sel_addr.

Return Value

The function return zero if it has successfully transferred the data, non-zero otherwise (e.g., if the address in offset is outside the limits of the segment whose selector is sel).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

readdir

Syntax

```
#include <dirent.h>

struct dirent *readdir(DIR *dir);
```

Description

This function reads entries from a directory opened by `opendir` (See opendir). It returns the information in a static buffer with this format:

```
struct dirent {
unsigned short d_namlen; /* The length of the name (like strlen) */
char d_name[MAXNAMLEN+1]; /* The name */
mode_t d_type; /* The file's type */
};
```

Note that some directory entries might be skipped by `readdir`, depending on the bits set in the global variable `__opendir_flags`. See opendir, __opendir_flags, opendir.

The possible values of the `d_type` member are:

`DT_REG`
: This is a regular file.

`DT_BLK`
: The file is a block device.

`DT_CHR`
: The file is a character device.

`DT_DIR`
: The file is a directory.

`DT_FIFO`
: This is a pipe (never happens in DJGPP).

`DT_LABEL`
: The file is a volume label.

`DT_LNK`
: The file is a symlink.

`DT_SOCK`
: The file is a socket.

`DT_UNKNOWN`
: The file's type is unknown. This value is put into the `d_type` member if the exact file's type is too expensive to compute. If the `__OPENDIR_NO_D_TYPE` flag is set in the global variable `__opendir_flags`, *all* files get marked with `DT_UNKNOWN`.

The macro `DTTOIF` (See DTTOIF) can be used to convert the `d_type` member to the equivalent value of the `st_mode` member of `struct stat`, see See stat.

Return Value

A pointer to a static buffer that is overwritten with each call.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. The `__opendir_flags` variable is DJGPP-specific. The `d_type` member is an extension available on some systems such as GNU/Linux.

Example

```
DIR *d = opendir(".");
struct dirent *de;
while (de = readdir(d))
puts(de->d_name);
closedir(d);
```

readlink

Syntax

```
#include <unistd.h>

int readlink(const char *filename, char *buffer, size_t size);
```

Description

MSDOS doesn't support symbolic links but DJGPP emulates them. This function checks if filename is a DJGPP symlink and the file name that the links points to is copied into buffer, up to maximum size characters. Portable applications should not assume that buffer is terminated with `'\0'`.

Return Value

Number of copied characters; value -1 is returned in case of error and `errno` is set. When value returned is equal to size, you cannot determine if there was enough room to copy whole name. So increase size and try again.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[FILENAME_MAX + 1];
if (readlink("/dev/env/DJDIR/bin/sh.exe", buf, FILENAME_MAX) == -1)
if (errno == EINVAL)
puts("/dev/env/DJDIR/bin/sh.exe is not a symbolic link.");
```

realloc

Syntax

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Description

This function changes the size of the region pointed to by ptr. If it can, it will reuse the same memory space, but it may have to allocate a new memory space to satisfy the request. In either case, it will return the pointer that you should use to refer to the (possibly new) memory area. The pointer passed may be `NULL`, in which case this function acts just like `malloc` (See malloc).

An application that wants to be robust in the face of a possible failure of `realloc` to enlarge a buffer should save a copy of the old pointer in a local variable, to be able to use the original buffer in case `realloc` returns `NULL`. See the example below for details.

Return Value

On success, a pointer is returned to the memory you should now refer to. On failure, `NULL` is returned and the memory pointed to by ptr prior to the call is not freed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (now+new > max)
{

char *old = p;

max = now+new;
p = realloc(p, max);
if (p == NULL)
p = old; /* retain the old pointer */
}
```

realpath

Syntax

```
#include <stdlib.h>

void realpath(const char *in_path, char *out_path);
```

Description

This function canonicalizes the input path in_path and stores the result in the buffer pointed to by out_path.

The path is canonicialized by removing consecutive and trailing slashes, making the path absolute if it's relative by prepending the current drive letter and working directory, removing "." components, collapsing ".." components, adding a drive specifier if needed, and converting all slashes to '/'. DOS-style 8+3 names of directories which are part of the pathname, as well as its final filename part, are returned lower-cased in out_path, but long filenames are left intact. See _preserve_fncase, for more details on letter-case conversions in filenames.

Since the returned path name can be longer than the original one, the caller should ensure there is enough space in the buffer pointed to by out_path. Use of ANSI-standard constant `FILENAME_MAX` (defined on `stdio.h`) or Posix-standard constant `PATH_MAX` (defined on `limits.h`) is recommended.

Return Value

If successful, a pointer to the result buffer is returned. Otherwise, `NULL` is returned and `errno` is set to indicate which error was detected.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
char oldpath[100], newpath[PATH_MAX];
scanf("%s", oldpath);
realpath(oldpath, newpath);
printf("that really is %s\n", newpath);
```

redir_cmdline_delete

Syntax

```
#include <debug/redir.h>

void redir_cmdline_delete (cmdline_t *cmd);
```

Description

For the rationale and general description of the debugger redirection issue, see See redir_debug_init.

This function serves as a destructor for a `cmdline_t` object. It frees storage used for the command-line arguments associated with cmd, closes any open handles stored in it, and frees memory used to store the file handles and the file names of the files where standard handles were redirected.

The function is safe to use even if cmd might be a `NULL` pointer, or if some of members of the `cmdline_t` structure are `NULL` pointers. See redir_debug_init, for detailed description of the `cmdline_t` structure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
redir_cmdline_delete (&child_cmd);
```

redir_cmdline_parse

Syntax

```
#include <debug/redir.h>

int redir_cmdline_parse (const char *args, cmdline_t *cmd);
```

Description

For the rationale and general description of the debugger redirection issue, see See redir_debug_init.

This function parses a command-line tail (i.e., without the program to be invoked) passed as a string in args. For every redirection directive in args, like `>> foo`, it opens the file that is the target of the redirection, and records in cmd the information about these redirections. (See redir_debug_init, for details of the `cmdline_t` structure that is used to hold this information.) The command line with redirections removed is placed into `cmd->command` (typically, it will be used to call `v2loadimage`, See v2loadimage), while the rest of information is used by `redir_to_child` and `redir_to_debugger` to redirect standard handles before and after calling `run_child`.

Return Value

The function returns zero in case of success, -1 otherwise. Failure usually means some kind of syntax error, like `>` without a file name following it; or a file name that isn't allowed by the underlying OS, like `lost+found` on DOS.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Init command line storage. */
if (redir_debug_init (&child_cmd) == -1)
fatal ("Cannot allocate redirection storage: not enough memory.\n");

/* Parse the command line and create redirections. */
if (strpbrk (args, "<>"))
{
if (redir_cmdline_parse (args, &child_cmd) == 0)
args = child_cmd.command;
else
error ("Syntax error in command line.");
}
else
child_cmd.command = strdup (args);

cmdline = (char *) alloca (strlen (args) + 4);
cmdline[0] = strlen (args);
strcpy (cmdline + 1, args);
cmdline[strlen (args) + 1] = 13;

if (v2loadimage (exec_file, cmdline, start_state))
{
printf ("Load failed for image %s\n", exec_file);
exit (1);
}
```

redir_debug_init

Syntax

```
#include <debug/redir.h>

int redir_debug_init (cmdline_t *cmd);
```

Description

This function initializes the data structure in the cmd variable required to save and restore debugger's standard handles across invocations of `run_child` (See run_child). The debugger will then typically call `redir_to_child` and `redir_to_debugger`.

These functions are needed when a debugger wants to redirect standard handles of the debuggee, or if the debuggee redirects some of its standard handles, because the debuggee is not a separate process, we just pretend it is by jumping between two threads of execution. But, as far as DOS is concerned, the debugger and the debuggee are a single process, and they share the same Job File Table (JFT). The JFT is a table maintained by DOS in the program's PSP where, for each open handle, DOS stores the index into the SFT, the System File Table. (The SFT is an internal data structure where DOS maintains everything it knows about a certain open file/device.) A handle that is returned by `open`, `_open` and other similar functions is simply an index into the JFT where DOS stored the SFT entry index for the file or device that the program opened.

When a program starts, the first 5 entries in the JFT are preconnected to the standard devices. Any additional handles opened by either the debugger or the debuggee use handles beyond the first 5 (unless one of the preconnected handles is deliberately closed). Here we mostly deal with handles 0, 1 and 2, the standard input, standard output, and standard error; they all start connected to the console device (unless somebody redirects the debugger's I/O from the command line).

Since both the debugger and the debuggee share the same JFT, their handles 0, 1 and 2 point to the same JFT entries and thus are connected to the same files/devices. Therefore, if the debugger redirects its standard output, the standard output of the debuggee is also automagically redirected to the same file/device! Similarly, if the debuggee redirects its stdout to a file, you won't be able to see debugger's output (it will go to the same file where the debuggee has its output); and if the debuggee closes its standard input, you will lose the ability to talk to debugger!

The debugger redirection support attempts to solve all these problems by creating an illusion of two separate sets of standard handles. Each time the debuggee is about to be run or resumed, it should call `redir_to_child` to redirect debugger's own standard handles to the file specified in the command-line (as given by e.g. the "run" command of GDB) before running the debuggee, then call `redir_to_debugger` to redirect them back to the debugger's original input/output when the control is returned from the debuggee (e.g. after a breakpoint is hit). Although the debugger and the debuggee have two separate copies of the file-associated data structures, the debugger still can redirect standard handles of the debuggee because they use the same JFT entries as debugger's own standard handles.

The `cmdline_t` structure is declared in the header `debug/redir.h` as follows:

```
struct dbg_redirect {
int inf_handle; /* debuggee's handle */
int our_handle; /* debugger's handle */
char *file_name; /* file name where debuggee's handle is
* redirected */
int mode; /* mode used to open() the above file */
off_t filepos; /* file position of debuggee's handle; unused */
};

typedef struct _cmdline {
char *command; /* command line with redirection removed */
int redirected; /* 1 if handles redirected for child */
struct dbg_redirect **redirection;/* info about redirected handles */
} cmdline_t;
```

In the `cmdline_t` structure, the `redirection` member points to an array of 3 `dbg_redirect` structures, one each for each one of the 3 standard handles. The `inf_handle` and `our_handle` members of those structures are used to save the handle used, respectively, by the debuggee (a.k.a. the inferior process) and by the debugger.

The cmd variable is supposed to be defined by the debugger's application code. `redir_debug_init` is called to initialize that variable. It calls `redir_cmdline_delete` to close any open handles held in cmd and to free any allocated storage; then it fills cmd with the trivial information (i.e., every standard stream is connected to the usual handles 0, 1, and 2).

Return Value

`redir_debug_init` returns zero in case of success, or -1 otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (redir_debug_init (&child_cmd) == -1)
fatal ("Cannot allocate redirection storage: not enough memory.\n");
```

redir_to_child

Syntax

```
#include <debug/redir.h>

int redir_to_child (cmdline_t *cmd);
```

Description

For the rationale and general description of the debugger redirection issue, see See redir_debug_init.

This function redirects all 3 standard streams so that they point to the files/devices where the child (a.k.a. debuggee) process connected them. All three standard handles point to the console device by default, but this could be changed, either because the command line for the child requested redirection, like in `prog > foo`, or because the child program itself redirected one of its standard handles e.g. with a call to `dup2`.

`redir_to_child` uses information stored in the `cmdline_t` variable pointed to by the cmd argument to redirect the standard streams as appropriate for the debuggee, while saving the original debugger's handles to be restored by `redir_to_debugger`.

Return Value

The function returns zero in case of success, -1 in case of failure. Failure usually means the process has run out of available file handles.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
errno = 0;
if (redir_to_child (&child_cmd) == -1)
{
redir_to_debugger (&child_cmd);
error ("Cannot redirect standard handles for program: %s.",
strerror (errno));
}
```

redir_to_debugger

Syntax

```
#include <debug/redir.h>

int redir_to_debugger (cmdline_t *cmd);
```

Description

For the rationale and general description of the debugger redirection issue, see See redir_debug_init.

This function redirects all 3 standard streams so that they point to the files/devices where the debugger process connected them. All three standard handles point to the console device by default, but this could be changed, either because the command line for the child requested redirection, like in `prog > foo`, or because the child program itself redirected one of its standard handles e.g. with a call to `dup2`.

`redir_to_debugger` uses information stored in the `cmdline_t` variable pointed to by the cmd argument to redirect the standard streams as appropriate for the debugger, while saving the original debuggee's handles to be restored by `redir_to_child`.

Return Value

The function returns zero in case of success, -1 in case of failure. Failure usually means the process has run out of available file handles.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* Restore debugger's standard handles. */
errno = 0;
if (redir_to_debugger (&child_cmd) == -1)
error ("Cannot redirect standard handles for debugger: %s.",
strerror (errno));
```

regcomp

Syntax

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);
```

Description

This function is part of the implementation of POSIX 1003.2 regular expressions (REs).

`regcomp` compiles the regular expression contained in the pattern string, subject to the flags in cflags, and places the results in the `regex_t` structure pointed to by preg. (The regular expression syntax, as defined by POSIX 1003.2, is described below.)

The parameter cflags is the bitwise OR of zero or more of the following flags:

`REG_EXTENDED`
: Compile modern (extended) REs, rather than the obsolete (basic) REs that are the default.

`REG_BASIC`
: This is a synonym for 0, provided as a counterpart to `REG_EXTENDED` to improve readability. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

`REG_NOSPEC`
: Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the RE in pattern is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. `REG_EXTENDED` and `REG_NOSPEC` may not be used in the same call to `regcomp`.

`REG_ICASE`
: Compile for matching that ignores upper/lower case distinctions. See the description of regular expressions below for details of case-independent matching.

`REG_NOSUB`
: Compile for matching that need only report success or failure, not what was matched.

`REG_NEWLINE`
: Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, `[^` bracket expressions and `.` never match newline, a `^` anchor matches the null string after any newline in the string in addition to its normal function, and the `$` anchor matches the null string before any newline in the string in addition to its normal function.

`REG_PEND`
: The regular expression ends, not at the first NUL, but just before the character pointed to by the `re_endp` member of the structure pointed to by preg. The `re_endp` member is of type `const char *`. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, `regcomp` returns 0 and fills in the structure pointed to by preg. One member of that structure (other than `re_endp`) is publicized: `re_nsub`, of type `size_t`, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the `REG_NOSUB` flag was used).

Note that the length of the RE does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

Return Value

If `regcomp` succeeds, it returns zero; if it fails, it returns a non-zero error code, which is one of these:

`REG_BADPAT`
: invalid regular expression

`REG_ECOLLATE`
: invalid collating element

REG_ECTYPE
invalid character class

REG_EESCAPE
\ applied to unescapable character

REG_ESUBREG
invalid backreference number (e.g., larger than the number of parenthesized subexpressions in the RE)

REG_EBRACK
brackets [ ] not balanced

REG_EPAREN
parentheses ( ) not balanced

REG_EBRACE
braces { } not balanced

REG_BADBR
invalid repetition count(s) in { }

REG_ERANGE
invalid character range in [ ]

REG_ESPACE
ran out of memory (an RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}'` will eventually run almost any existing machine out of swap space)

REG_BADRPT
?, *, or + operand invalid

REG_EMPTY
empty (sub)expression

REG_ASSERT
''can't happen'' (you found a bug in `regcomp`)

REG_INVARG
invalid argument (e.g. a negative-length string)

Regular Expressions' Syntax

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of `egrep`; 1003.2 calls these *extended* REs) and obsolete REs (roughly those of `ed`; 1003.2 *basic* REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. 1003.2 leaves some aspects of RE syntax and semantics open; '(*)' marks decisions on these aspects that may not be fully portable to other 1003.2 implementations.

A (modern) RE is one(*) or more non-empty(*) *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is one(*) or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single(*) `*`, `+`, `?`, or *bound*. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a sequence of 0 or 1 matches of the atom.

A *bound* is `{` followed by an unsigned decimal integer, possibly followed by `,` possibly followed by another unsigned decimal integer, always followed by `}`. The integers must lie between 0 and `RE_DUP_MAX` (255(*)) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer `i` and no comma matches a sequence of exactly `i` matches of the atom. An atom followed by a bound containing one integer `i` and a comma matches a sequence of `i` or more matches of the atom. An atom followed by a bound containing two integers `i` and `j` matches a sequence of `i` through `j` (inclusive) matches of the atom.

An atom is a regular expression enclosed in `()` (matching a match for the regular expression), an empty set of `()` (matching the null string(*)), a *bracket expression* (see below), `.` (matching any single character), `^` (matching the null string at the beginning of a line), $ (matching the null string at the end of a line), a \ followed by one of the characters ^.[$()|*+?{\\ (matching that character taken as an ordinary character), a \ followed by any other character(*) (matching that character taken as an ordinary character, as if the \ had not been present(*)), or a single character with no other significance (matching that character). A { followed by a character other than a digit is an ordinary character, not the beginning of a bound(*). It is illegal to end an RE with \.

A *bracket expression* is a list of characters enclosed in []. It normally matches any single character from the list (but see below). If the list begins with ^, it matches any single character (but see below) *not* from the rest of the list. If two characters in the list are separated by -, this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence, e.g. [0-9] in ASCII matches any decimal digit. It is illegal(*) for two ranges to share an endpoint, e.g. a-c-e. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal] in the list, make it the first character (following a possible ^). To include a literal -, make it the first or last character, or the second endpoint of a range. To use a literal - as the first endpoint of a range, enclose it in [. and .] to make it a collating element (see below). With the exception of these and some combinations using [(see next paragraphs), all other special characters, including \, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in [. and .] stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression's list. A bracket expression containing a multi-character collating element can thus match more than one character, e.g. if the collating sequence includes a ch collating element, then the RE [[.ch.]]*c matches the first five characters of ''chchcc''.

Within a bracket expression, a collating element enclosed in [= and =] is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were [. and .].) For example, if o and ^ are the members of an equivalence class, then [[=o=]], [[=^=]], and [o^] are all synonymous. An equivalence class may *not* be an endpoint of a range.

Within a bracket expression, the name of a *character class* enclosed in [: and :] stands for the list of all characters belonging to that class. Standard character class names are:

```
alnum digit punct
alpha graph space
blank lower upper
cntrl print xdigit
```

These stand for the character classes defined by isalnum (See isalnum), isdigit (See isdigit), ispunct (See ispunct), isalpha (See isalpha), isgraph (See isgraph), isspace (See isspace) (blank is the same as space), islower (See islower), isupper (See isupper), iscntrl (See iscntrl), isprint (See isprint), and isxdigit (See isxdigit), respectively. A locale may provide others. A character class may not be used as an endpoint of a range.

There are two special cases(*) of bracket expressions: the bracket expressions [[:<:]] and [[:>:]] match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an alnum character (as defined by isalnum library function) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, bb* matches the three middle characters of ''abbbc'', (wee|week)(knights|nights) matches all ten characters of ''weeknights'', when (.*).* is matched against ''abc'' the parenthesized subexpression matches all three characters, and when (a*)* is matched against ''bc'' both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. x becomes [xX]. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

No particular limit is imposed on the length of REs(*). Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete (*basic*) regular expressions differ in several respects. `|`, `+`, and `?` are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. `^` is an ordinary character except at the beginning of the RE or(*) the beginning of a parenthesized subexpression, `$` is an ordinary character except at the end of the RE or(*) the end of a parenthesized subexpression, and `*` is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^`). Finally, there is one new type of atom, a *back reference*: `\` followed by a non-zero decimal digit *d* matches the same sequence of characters matched by the *d*th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) `\([bc]\)\1` matches ‘‘bb’’ or ‘‘cc’’ but not ‘‘bc’’.

History

This implementation of the POSIX regexp functionality was written by henry@zoo.toronto.edu, Henry Spencer.

Bugs

The locale is always assumed to be the default one of 1003.2, and only the collating elements etc. of that locale are available.

`regcomp` implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested.

An RE like, say, `((((a{1,100}){1,100}){1,100}){1,100}){1,100}`, will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like `a)b` are legal REs because `)` is a special character only in the presence of a previous unmatched `(`. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does `a\e(\e(b\e)*\e2\e)*d` match ‘‘abbbd’’? Until the standard is clarified, behavior in such cases should not be relied on.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

regerror

Syntax

```
#include <sys/types.h>
#include <regex.h>

size_t regerror(int errcode, const regex_t *preg,
char *errbuf, size_t errbuf_size);
```

Description

`regerror` maps a non-zero value of errcode from either `regcomp` (Return Value, See regcomp) or `regexec` (Return Value, See regexec) to a human-readable, printable message.

If preg is non-`NULL`, the error code should have arisen from use of the variable of the type `regex_t` pointed to by preg, and if the error code came from `regcomp`, it should have been the result from the most recent `regcomp` using that `regex_t` variable. (`regerror` may be able to supply a more detailed message using information from the `regex_t` than from errcode alone.) `regerror` places the `NUL`-terminated message into the buffer pointed to by errbuf, limiting the length (including the `NUL`) to at most errbuf_size bytes. If the whole message won’t fit, as much of it as will fit before the terminating `NUL` is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating `NUL`). If errbuf_size is 0, errbuf is ignored but the return value is still correct.

If the errcode given to `regerror` is first ORed with `REG_ITOA`, the ''message'' that results is the printable name of the error code, e.g. ''REG_NOMATCH'', rather than an explanation thereof. If errcode is `REG_ATOI`, then preg shall be non-NULL and the `re_endp` member of the structure it points to must point to the printable name of an error code (e.g. ''REG_ECOLLATE''); in this case, the result in errbuf is the decimal representation of the numeric value of the error code (0 if the name is not recognized). `REG_ITOA` and `REG_ATOI` are intended primarily as debugging facilities; they are extensions, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

Return Value

The size of buffer needed to hold the message (including terminating `NUL`) is always returned, even if errbuf_size is zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

regexec

Syntax

```
#include <sys/types.h>
#include <regex.h>

int regexec(const regex_t *preg, const char *string,
size_t nmatch, regmatch_t pmatch[], int eflags);
```

Description

`regexec` matches the compiled RE pointed to by preg against the string, subject to the flags in eflags, and reports results using nmatch, pmatch, and the returned value. The RE must have been compiled by a previous invocation of `regcomp` (See regcomp). The compiled form is not altered during execution of `regexec`, so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by string is considered to be the text of an entire line, with the NUL indicating the end of the line. (That is, any other end-of-line marker is considered to have been removed and replaced by the NUL.)

The eflags argument is the bitwise OR of zero or more of the following flags:

`REG_NOTBOL`
: The first character of the string is not the beginning of a line, so the `^` anchor should not match before it. This does not affect the behavior of newlines under `REG_NEWLINE` (REG_NEWLINE, See regcomp).

`REG_NOTEOL`
: The NUL terminating the string does not end a line, so the `$` anchor should not match before it. This does not affect the behavior of newlines under `REG_NEWLINE` (REG_NEWLINE, See regcomp).

`REG_STARTEND`
: The string is considered to start at `string + pmatch[0].rm_so` and to have a terminating `NUL` located at `string + pmatch[0].rm_eo` (there need not actually be a `NUL` at that location), regardless of the value of nmatch. See below for the definition of pmatch and nmatch. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero `rm_so` does not imply `REG_NOTBOL`; `REG_STARTEND` affects only the location of the string, not how it is matched.

`REG_TRACE`
: trace execution (printed to stdout)

`REG_LARGE`
: force large representation

`REG_BACKR`
: force use of backref code

Regular Expressions' Syntax, See regcomp, for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of string.

If REG_NOSUB was specified in the compilation of the RE (REG_NOSUB, See regcomp), or if nmatch is 0, regexec ignores the pmatch argument (but see below for the case where REG_STARTEND is specified). Otherwise, pmatch should point to an array of nmatch structures of type regmatch_t. Such a structure has at least the members rm_so and rm_eo, both of type regoff_t (a signed arithmetic type at least as large as an off_t and a ssize_t), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the string argument given to regexec. An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

When regexec returns, the 0th member of the pmatch array is filled in to indicate what substring of string was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member i reports subexpression i, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array---corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, i > preg->re_nsub)---have both rm_so and rm_eo set to -1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE (b*)+ matches ‘‘bbb’’, the parenthesized subexpression matches the three bs and then an infinite number of empty strings following the last b, so the reported substring is one of the empties.)

If REG_STARTEND is specified in eflags, pmatch must point to at least one regmatch_t variable (even if nmatch is 0 or REG_NOSUB was specified in the compilation of the RE, REG_NOSUB, See regcomp), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by nmatch; if nmatch is 0 or REG_NOSUB was specified, the value of pmatch[0] will not be changed by a successful regexec.

Return Value

Normally, regexec returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations. The list of possible error return values is below:

REG_ESPACE
: ran out of memory

REG_BADPAT
: the passed argument preg doesn’t point to an RE compiled by regcomp

REG_INVARG
: invalid argument(s) (e.g., string + pmatch[0].rm_eo is less than string + pmatch[0].rm_so)

History

This implementation of the POSIX regexp functionality was written by henry@zoo.toronto.edu, Henry Spencer.

Bugs

regexec performance is poor. nmatch exceeding 0 is expensive; nmatch exceeding 1 is worse. regexec is largely insensitive to RE complexity *except* that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

regfree

Syntax

```
#include <sys/types.h>
#include <regex.h>

void regfree(regex_t *preg);
```

Description

`regfree` frees any dynamically-allocated storage associated with the compiled RE pointed to by preg. The remaining `regex_t` is no longer a valid compiled RE and the effect of supplying it to `regexec` or `regerror` is undefined.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

remove

Syntax

```
#include <stdio.h>

int remove(const char *file);
```

Description

This function removes the named file from the file system. Unless you have an un-erase program, the file and its contents are gone for good.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
remove("/tmp/data.tmp");
```

remque

Syntax

```
#include <search.h>

void remque(struct qelem *elem);
```

Description

This function manipulates queues built from doubly linked lists. Each element in the queue must be in the form of `struct qelem` which is defined thus:

```
struct qelem {
struct qelem *q_forw;
struct qelem *q_back;
char q_data[0];
}
```

This function removes the entry elem from a queue.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_rename

Syntax

```
#include <stdio.h>

int _rename(const char *oldname, const char *newname);
```

Description

This function renames an existing file or directory oldname to newname. It is much smaller that `rename` (See rename), but it can only rename a directory so it stays under the same parent, it cannot move directories between different branches of the directory tree. This means that in the following example, the first call will succeed, while the second will fail:

```
_rename("c:/path1/mydir", "c:/path1/yourdir");
_rename("c:/path1/mydir", "c:/path2");
```

On systems that support long filenames (See _use_lfn), `_rename` can also move directories (so that both calls in the above example succeed there), unless the `LFN` environment variable is set to **n**, or the `_CRT0_FLAG_NO_LFN` is set in the `_crt0_startup_flags` variable, See _crt0_startup_flags.

If you don't need the extra functionality offered by `rename` (which usually is only expected by Unix-born programs), you can use `_rename` instead and thus make your program a lot smaller.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

rename

Syntax

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

Description

This function renames an existing file or directory oldname to newname. If newname exists, then it is first removed. If newname is a directory, it must be empty (or else `errno` will be set to `ENOTEMPTY`), and must not include oldname in its path prefix (otherwise, `errno` will be set to `EINVAL`). If newname exists, both oldname and newname must be of the same type (both directories or both regular files) (or else `errno` will be set to `ENOTDIR` or `EISDIR`), and must reside on the same logical device (otherwise, `errno` will be set to `EXDEV`). Wildcards are not allowed in either oldname or newname. DOS won't allow renaming a current directory even on a non-default drive (you will get the `EBUSY` or `EINVAL` in `errno`). `ENAMETOOLONG` will be returned for pathnames which are longer than the limit imposed by DOS. If oldname doesn't exist, `errno` will be set to `ENOENT`. For most of the other calamities, DOS will usually set `errno` to `EACCES`.

If anything goes wrong during the operation of `rename()`, the function tries very hard to leave the things as ther were before it was invoked, but it might not always succeed.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
rename("c:/mydir/some.doc", "c:/yourdir/some.sav");
rename("c:/path1/mydir", "c:/path2");
```

rewind

Syntax

```
#include <stdio.h>

void rewind(FILE *file);
```

Description

This function repositions the file pointer to the beginning of the file and clears the error indicator.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
rewind(stdin);
```

rewinddir

Syntax

```
#include <dirent.h>

void rewinddir(DIR *dir);
```

Description

This function resets the position of the dir so that the next call to `readdir` (See readdir) starts at the beginning again.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
DIR *d = opendir(".");
rewinddir(d);
```

rindex

Syntax

```
#include <strings.h>

char *rindex(const char *string, int ch);
```

Description

Returns a pointer to the last occurrence of ch in string. Note that the `NULL` character counts, so if you pass zero as ch you'll get a pointer to the end of the string back.

Return Value

A pointer to the character, or `NULL` if it wasn't found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *last_slash = rindex(filename, ’/’);
```

rmdir

Syntax

```
#include <unistd.h>

int rmdir(const char *dirname);
```

Description

This function removes directory dirname. The directory must be empty.

Return Value

Zero if the directory was removed, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
rmdir("/tmp/datadir");
```

run_child

Syntax

```
#include <debug/dbgcom.h>

void run_child (void);
```

Description

This function starts or resumes the debugged program, via a `longjmp` to the debuggee's code. When the debuggee hits a breakpoint, or exits normally, the exception handler that is called to service the breakpoint exception will `longjmp` back to `run_child`, and it will then return to the caller.

After `run_child` returns, the debugger usually examines the `a_tss` variable to find out the reason the debuggee stopped. The `a_tss` variable is defined by the header `debug/tss.h` as follows:

```
typedef struct TSS {
unsigned short tss_back_link;
unsigned short res0;
unsigned long tss_esp0;
unsigned short tss_ss0;
unsigned short res1;
unsigned long tss_esp1;
unsigned short tss_ss1;
unsigned short res2;
unsigned long tss_esp2;
unsigned short tss_ss2;
unsigned short res3;
unsigned long tss_cr3;

unsigned long tss_eip;
unsigned long tss_eflags;
unsigned long tss_eax;
unsigned long tss_ecx;
unsigned long tss_edx;
unsigned long tss_ebx;
unsigned long tss_esp;
unsigned long tss_ebp;
unsigned long tss_esi;
unsigned long tss_edi;
unsigned short tss_es;
unsigned short res4;
unsigned short tss_cs;
unsigned short res5;
unsigned short tss_ss;
unsigned short res6;
unsigned short tss_ds;
```

```
unsigned short res7;
unsigned short tss_fs;
unsigned short res8;
unsigned short tss_gs;
unsigned short res9;
unsigned short tss_ldt;
unsigned short res10;
unsigned short tss_trap;
unsigned char tss_iomap;
unsigned char tss_irqn;
unsigned long tss_error;
} TSS;

extern TSS a_tss;
```

See the example below for a typical tests after `run_child` returns.

Note that, generally, you'd need to save the standard handles before calling `run_child` and restore them after it returns. Otherwise, if the debuggee redirects one of its standard handles, the corresponding debugger's standard handle is redirected as well. See redir_to_child, and see See redir_to_debugger.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
save_npx ();
run_child ();
load_npx ();

if (a_tss.tss_irqn == 0x21)
{
status = DEBUGGEE_EXITED;
exit_code = a_tss.tss_eax & 0xff;
}
else
{
status = DEBUGGEE_GOT_SIGNAL
if (a_tss.tss_irqn == 0x75)
signal_number = SIGINT;
else if (a_tss.tss_irqn == 1 || a_tss.tss_irqn == 3)
signal_number = SIGTRAP; /* a breakpoint */
}
```

save_npx

Syntax

```
#include <debug/dbgcom.h>

extern NPX npx;
void save_npx (void);
```

Description

This function saves the state of the x87 numeric processor in the external variable `npx`. This variable is a structure defined as follows in the header `debug/dbgcom.h`:

```
typedef struct {
unsigned short sig0;
unsigned short sig1;
unsigned short sig2;
unsigned short sig3;
unsigned short exponent:15;
unsigned short sign:1;
} NPXREG;

typedef struct {
```

```
unsigned long control;
unsigned long status;
unsigned long tag;
unsigned long eip;
unsigned long cs;
unsigned long dataptr;
unsigned long datasel;
NPXREG reg[8];
long double st[8];
char st_valid[8];
long double mmx[8];
char in_mmx_mode;
char top;
} NPX;
```

`save_npx` should be called immediately before `run_child` (See run_child) is called to begin or resume the debugged program.

To restore the x87 state when control is returned to the debugger, call `load_npx`, see See load_npx.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
save_npx ();
run_child ();
load_npx ();
```

sbrk

Syntax

```
#include <unistd.h>

void *sbrk(int delta)
```

Description

This function changes the "break" of the program by adding delta to it. This is the highest address that your program can access without causing a violation. Since the heap is the region under the break, you can expand the heap (where `malloc` gets memory from) by increasing the break.

This function is normally accessed only by `malloc` (See malloc).

Return Value

The address of the first byte outside of the previous valid address range, or -1 if no more memory could be accessed. In other words, a pointer to the chunk of heap you just allocated, if you had passed a positive number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *buf;
buf = sbrk(1000); /* allocate space */
```

scanf

Syntax

```
#include <stdio.h>

int scanf(const char *format, ...);
```

Description

This function scans formatted text from `stdin` and stores it in the variables pointed to by the arguments. See scanf.

The format string contains regular characters which much match the input exactly as well as a conversion specifiers, which begin with a percent symbol. Any whitespace in the format string matches zero or more of any whitespace characters in the input. Thus, a single space may match a newline and two tabs in the input. All conversions except `c` and `[` also skip leading whitespace automatically. Each conversion specifier contains the following fields:

- An asterisk (*) which indicates that the input should beconverted according to the conversion spec, but not stored anywhere. This allows to describe an input field that is to be skipped.
- A width specifier, which specifies the maximum number of inputcharacters to use in the conversion.
- An optional conversion qualifier, which may be:

  `hh`
  : to specify `char`;

  `h`
  : to specify `short` integers;

  `j`
  : to specify `intmax_t` or `uintmax_t` integers;

  `l`
  : to specify doubles or `long` integers;

  `ll`
  : (two lower-case ell letters) to specify `long long` integers; to specify `long` doubles, although this is non-standard;

  `L`
  : to specify `long` doubles;

  `t`
  : to specify `ptrdiff_t`;

  `z`
  : to specify `size_t`.

  If the `h` qualifier appears before a specifier that implies conversion to a `long` or `float` or `double`, like in `%hD` or `%hf`, it is generally ignored.

- The conversion type specifier*Some of the combinations*listed below are non-standard. If you use the non-standard specifiers, a compiler could complain.:

  `c`
  : Copy the next character (or width characters) to the given buffer. This conversion suppresses skipping of the leading whitespace; use `%1s` to read the next non-whitespace character. Unlike with `%s`, the copied characters are **not** terminated with a null character. If the width parameter is not specified, a width of one is implied.

  `d`
  : Convert the input to a signed `int` using 10 as the base of the number representation.

  `hhd`
  : Convert the input to a signed `char` using 10 as the base.

  `hd`
  : Convert the input to a signed `short` using 10 as the base.

  `jd`
  : Convert the input to an `intmax_t` using 10 as the base.

  `ld`
  `D`
  : Convert the input to a signed `long` using 10 as the base.

  `Ld`

`lld`
`lD`
: Convert the input to a signed `long long` using 10 as the base.

`td`
: Convert the input to a `ptrdiff_t` using 10 as the base.

`zd`
: Convert the input to a `size_t` using 10 as the base.

`e`
`E`
`f`
`F`
`g`
`G`
: Convert the input to a floating point number (a `float`).

`le`
`lE`
`lf`
`lF`
`lg`
`lG`
: Convert the input to a `double`.

`Le`
`LE`
`lle`
`llE`
`Lf`
`LF`
`llf`
`llF`
`Lg`
`LG`
`llg`
`llG`
: Convert the input to a `long double`.

`i`
: Convert the input, determining base automatically by the presence of `0x` or `0` prefixes, and store in a signed `int`. See strtol.

`hhi`
: Like `i`, but stores the result in a signed `char`.

`hi`
: Like `i`, but stores the result in a signed `short`.

`ji`
: Like `i`, but stores the result in an `intmax_t`.

`li`
`I`
: Like `i`, but stores the result in a signed `long`.

`Li`
`lli`
`lI`
: Like `i`, but stores the result in a signed `long long`.

`ti`
: Like `i`, but stores the result in a `ptrdiff_t`.

`zi`
: Like `i`, but stores the result in a `size_t`.

n
: Store the number of characters scanned so far into the `int` pointed to by the argument.

hhn
: Like `n`, but the argument should point to a signed `char`.

hn
: Like `n`, but the argument should point to a signed `short`.

jn
: Like `n`, but the argument should point to an `intmax_t`.

ln
: Like `n`, but the argument should point to a signed `long`.

Ln
lln
: Like `n`, but the argument should point to a signed `long long`.

tn
: Like `n`, but the argument should point to a `ptrdiff_t`.

zn
: Like `n`, but the argument should point to a `size_t`.

o
: Convert the input to an unsigned `int`, using base 8.

hho
: Convert the input to an unsigned `char`, using base 8.

ho
: Convert the input to an unsigned `short`, using base 8.

jo
: Convert the input to an `uintmax_t`, using base 8.

lo
O
: Convert the input to an unsigned `long`, using base 8.

Lo
llo
lO
: Convert the input to an unsigned `long long`, using base 8.

to
: Convert the input to a `ptrdiff_t`, using base 8.

zo
: Convert the input to a `size_t`, using base 8.

p
: Convert the input to a pointer. This is like using the `x` format.

s
: Copy the input to the given string, skipping leading whitespace and copying non-whitespace characters up to the next whitespace. The string stored is then terminated with a null character.

u
: Convert the input to an unsigned `int` using 10 as the base.

hhu
: Convert the input to an unsigned `char` using 10 as the base.

hu
: Convert the input to an unsigned `short` using 10 as the base.

`ju`
: Convert the input to an `uintmax_t` using 10 as the base.

`lu`
`U`
: Convert the input to an unsigned `long` using 10 as the base.

`Lu`
`llu`
`lU`
: Convert the input to an unsigned `long long` using 10 as the base.

`tu`
: Convert the input to a `ptrdiff_t` using 10 as the base.

`zu`
: Convert the input to a `size_t` using 10 as the base.

`x`
`X`
: Convert the input to an unsigned `int`, using base 16.

`hhx`
`hhX`
: Convert the input to an unsigned `char`, using base 16.

`hx`
`hX`
: Convert the input to an unsigned `short`, using base 16.

`jx`
`jX`
: Convert the input to an `uintmax_t`, using base 16.

`lx`
`lX`
: Convert the input to an unsigned `long`, using base 16.

`Lx`
`LX`
`llx`
`llX`
: Convert the input to an unsigned `long long`, using base 16.

`tx`
`tX`
: Convert the input to a `ptrdiff_t`, using base 16.

`zx`
`zX`
: Convert the input to a `size_t`, using base 16.

`[…]`
: Stores the matched characters in a `char` array, followed by a terminating null character. If you do not specify the width parameter, `scanf` behaves as if width had a very large value. Up to width characters are consumed and assigned, provided that they match the specification inside the brackets. The characters between the brackets determine which characters are allowed, and thus when the copying stops. These characters may be regular characters (example: `[abcd]`) or a range of characters (example: `[a-d]`). If the first character is a caret (^), then the set specifies the set of characters that do not get copied (i.e. the set is negated). To specify that the set contains a close-bracket (]), put it immediately after `[` or `[^`. To specify a literal dash (-), write it either immediately after `[` or `[^`, or immediately before the closing `]`.

`%`
: This must match a percent character in the input.

Integer formats make use of `strtol` or `strtoul` to perform the actual conversions. Floating-point conversions use `strtod` and `_strtold`.

Return Value

The number of items successfully matched and assigned. If input ends, or if there is any input failure before the first item is converted and assigned, `EOF` is returned. Note that literal characters (including whitespace) in the format string which matched input characters count as ''converted items'', so input failure *after* such characters were read and matched will **not** cause `EOF` to be returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1) (see note 2) 1003.2-1992; 1003.1-2001

Notes:

1. The `hh`, `j`, `t` and `z` conversion specifiers first appeared in the ANSI C99 standard.

2. The conversion specifiers `F`, `D`, `I`, `O`, and `U` are DJGPP extensions; they are provided for compatibility with Borland C and other compilers. The conversion specifiers for the `long long` data type are GCC extensions. The meaning of `[a-c]` as a range of characters is a very popular extension to ANSI (which merely says a dash ''may have a special meaning'' in that context).

Example

```
int x, y;
char buf[100];
scanf("%d %d %s", &x, &y, buf);

/* read to end-of-line */
scanf("%d %[^\n]\n", &x, buf);
/* read letters only */
scanf("%[a-zA-Z]", buf);
```

SCN

Syntax

```
#include <inttypes.h>
```

Description

The `SCN` family of macros allows integers to be input in a portable manner using the `scanf` family of functions (See scanf). They include a conversion qualifier, to specify the width of the type (e.g.: `l` for `long`), and the conversion type specifier (e.g.: `d` for decimal display of integers).

The `SCN` family of macros should be used with the types defined in the header `<stdint.h>`. For example: `int8_t`, `uint_fast32_t`, `uintptr_t`, `intmax_t`.

Below N can be 8, 16, 32 or 64. The `SCN` macros are:

`SCNdN`
`SCNiN`
: The `d` and `i` type conversion specifiers for a type `intN_t` of N bits.

`SCNdLEASTN`
`SCNiLEASTN`
: The `d` and `i` type conversion specifiers for a type `int_leastN_t` of N bits.

`SCNdFASTN`
`SCNiFASTN`
: The `d` and `i` type conversion specifiers for a type `int_fastN_t` of N bits.

`SCNdMAX`
`SCNiMAX`
: The `d` and `i` type conversion specifiers for a type `intmax_t`.

`SCNdPTR`
`SCNiPTR`
: The `d` and `i` type conversion specifier for a type `intptr_t`.

`SCNoN`
`SCNuN`
`SCNxN`
The `o`, `u` and `x` type conversion specifiers for a type `uintN_t` of N bits.

`SCNoLEASTN`
`SCNuLEASTN`
`SCNxLEASTN`
The `o`, `u` and `x` type conversion specifiers for a type `uint_LEASTN_t` of N bits.

`SCNoFASTN`
`SCNuFASTN`
`SCNxFASTN`
The `o`, `u` and `x` type conversion specifiers for a type `uint_FASTN_t` of N bits.

`SCNoMAX`
`SCNuMAX`
`SCNxMAX`
The `o`, `u` and `x` type conversion specifiers for a type `uintmax_t`.

`SCNoPTR`
`SCNuPTR`
`SCNxPTR`
The `o`, `u` and `x` type conversion specifiers for a type `uintptr_t`.

Return Value

Not applicable.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
intmax_t m;
int ret;

ret = sscanf("0x1000", "%" SCNxMAX, &m);
```

Screen Variables

Syntax

```
#include <go32.h>
#include <pc.h>

unsigned long ScreenPrimary;
unsigned long ScreenSecondary;
extern unsigned char ScreenAttrib;
```

Description

The first two variables (actually, they are #define'd aliases to fields in the _go32_info_block structure See _go32_info_block) allow access to the video memory of the primary and secondary screens as if they were arrays. To reference them, you must use dosmemget()/dosmemput() functions (See dosmemget, See dosmemput) or any one of the far pointer functions (See _far*), as the video memory is *not* mapped into your default address space.

The variable ScreenAttrib holds the current attribute which is in use by the text screen writes. The attribute is constructed as follows:

bits 0-3 -- foreground color;

bits 4-6 -- background color;

bit 7 -- blink on (1) or off (0).

Example

_farpokew(_dos_ds, ScreenPrimary, (((unsigned short) attr) << 8) + char));

ScreenClear

Syntax

```
#include <pc.h>

void ScreenClear(void);
```

Description

This function clears the text screen. It overwrites it by blanks with the current background and foreground as specified by ScreenAttrib (See Screen Variables).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenClear();
```

ScreenCols

Syntax

```
#include <pc.h>

int ScreenCols(void);
```

Description

This function returns the number of columns of the screen. It does so by looking at the byte at the absolute address 40:4Ah in the BIOS area. In text modes, the meaning of number of columns is obvious; in graphics modes, this value is the number of columns of text available when using the video BIOS functions to write text.

Return Value

The number of columns.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int available_columns = ScreenCols();
```

ScreenGetChar

Syntax

```
#include <pc.h>

void ScreenGetChar(int *ch, int *attr, int col, int row);
```

Description

This function stores the character and attribute of the current primary screen at row given by row and column given by col (these are zero-based) into the integers whose address is specified by ch and attr. It does so by directly accessing the video memory, so it will only work when the screen is in text mode. You can pass the value `NULL` in each of the pointers if you do not want to retrieve the the corresponding information.

Warning: note that both the variables ch and attr are pointers to an `int`, not to a `char`! You **must** pass a pointer to an `int` there, or your program will crash or work erratically.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int ch, attr;

ScreenGetChar(&ch, &attr, 0, 0);
```

ScreenGetCursor

Syntax

```
#include <pc.h>

void ScreenGetCursor(int *row, int *column);
```

Description

This function retrieves the current cursor position of the default video page by calling function 3 of the interrupt 10h, and stores it in the variables pointed by row and column.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenGetCursor(&wherex, &wherey);
```

ScreenMode

Syntax

```
#include <pc.h>

int ScreenMode(void);
```

Description

This function reports the current video mode as known to the system BIOS. It does so by accessing the byte at absolute address 40:49h.

Return Value

The video mode.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
video_mode = ScreenMode();
```

ScreenPutChar

Syntax

```
#include <pc.h>

void ScreenPutChar(int ch, int attr, int col, int row);
```

Description

This function writes the character whose value is specified in ch with an attribute attr at row given by row and column given by col, which are zero-based. It does so by directly accessing the video memory, so it will only work when the screen is in text mode.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenPutChar('R', (BLUE << 4) | LIGHTMAGENTA, 75, 0);
```

ScreenPutString

Syntax

```
#include <pc.h>

void ScreenPutString(const char *str, int attr, int column, int row);
```

Description

Beginning at screen position given by column and row, this function displays the string given by str. Each string character gets the attribute given by attr. If column or row have values outside legal range for current video mode, nothing happens. The variables row and column are zero-based (e.g., the topmost row is row 0).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenPutString("Hello, world!", (BLUE << 4) | LIGHTBLUE, 20, 10);
```

ScreenRetrieve

Syntax

```
#include <pc.h>

void ScreenRetrieve(void *buf);
```

Description

This function stores a replica of the current primary screen contents in the buffer pointed to by buf. It assumes without checking that buf has enough storage to hold the data. The required storage can be computed as `ScreenRows()*ScreenCols()*2` (See ScreenRows, See ScreenCols).

Return Value

None.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
unsigned *saved_screen = (unsigned *)alloca(ScreenRows()*ScreenCols()*2;

ScreenRetrieve(saved_screen);
```

ScreenRows
Syntax
```
#include <pc.h>

int ScreenRows(void);
```

Description
This function returns the number of rows of the text screen. It does so by looking at the byte at the absolute address 40:84h in the BIOS area. This method works only for video adapters with their own BIOS extensions, like EGA, VGA, SVGA etc.

Return Value
The number of rows.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
int rows = ScreenRows();
```

ScreenSetCursor
Syntax
```
#include <pc.h>

void ScreenSetCursor(int row, int column);
```

Description
This function moves the cursor position on the default video page to the point given by (zero-based) row and column, by calling function 2 of interrupt 10h.

Return Value
None.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
ScreenSetCursor(0, 0); /* home the cursor */
```

ScreenUpdate
Syntax

```
#include <pc.h>

void ScreenUpdate(void *buf);
```

Description

This function writes the contents of the buffer buf to the primary screen. The buffer should contain an exact replica of the video memory, including the characters and their attributes.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenUpdate(saved_screen);
```

ScreenUpdateLine

Syntax

```
#include <pc.h>

void ScreenUpdateLine(void *buf, int row);
```

Description

This function writes the contents of buf to the screen line number given in row (the topmost line is row 0), on the primary screen.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenUpdateLine(line_buf, 10);
```

ScreenVisualBell

Syntax

```
#include <pc.h>

void ScreenVisualBell(void);
```

Description

This function flashes the screen colors to produce the effect of ‘‘visual bell’. It does so by momentarily inverting the colors of every character on the screen.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
ScreenVisualBell();
```

searchpath

Syntax

```
#include <dir.h>
char * searchpath(const char *file);
```

Description

Given a name of a file in file, searches for that file in a list of directories, including the current working directory and directories listed in the `PATH` environment variable, and if found, returns the file name with leading directories prepended, so that the result can be used to access the file (e.g. by calling `open` or `stat`).

If file includes a drive letter or leading directories, `searchpath` first tries that name unaltered, in case it is already a fully-qualified path, or is relative to the current working directory. If that fails, it tries every directory in `PATH` in turn. Note that this will find e.g. `c:/foo/bar/baz.exe` if you pass `bar/baz.exe` to `searchpath` and if `c:/foo` is mentioned in `PATH`.

Return Value

When successfull, the function returns a pointer to a static buffer where the full pathname of the found file is stored. Otherwise, it returns `NULL`. (The static buffer is overwritten on each call.)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

This function is provided for compatibility with Borland's library. However, note that the Borland version disregards the leading directories altogether and searches for the basename only. Thus, it will happily find e.g. `c:/foo/bar/baz.exe`, even if the directory `c:/foo/bar` doesn't exist, provided that `baz.exe` is somewhere on your `PATH`. We think this is a bug, so DJGPP's implementation doesn't behave like that.

Example

```
printf("%s was found as %s\n", argv[1], searchpath(argv[1]));
```

seekdir

Syntax

```
#include <dirent.h>

void seekdir(DIR *dir, long loc);
```

Description

This function sets the location pointer in dir to the specified loc. Note that the value used for loc should be either zero or a value returned by `telldir` (See telldir). The next call to `readdir` (See readdir) will read whatever entry follows that point in the directory.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int q = telldir(dir);
do_stuff();
seekdir(dir, q);
```

select

Syntax

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

int
```

```
select(int nfds,
fd_set *readfds,
fd_set *writefds,
fd_set *exceptfds,
struct timeval *timeout)
```

Description

This function waits for files to be ready for input or output, or to have exceptional condition pending, or for a timeout.

Each `fd_set` variable is a bitmap representation of a set of file descriptors, one bit for every descriptor. The following macros shall be used to deal with these sets (in the table below, p is a pointer to an `fd_set` object and n is a file descriptor):

`FD_ZERO(p)`
: Initialize the set p to all zeros.

`FD_SET(n, p)`
: Set member n in set p.

`FD_CLR(n, p)`
: Clear member n in set p.

`FD_ISSET(n, p)`
: Return the value of member n in set p.

`FD_SETSIZE`
: The maximum number of descriptors supported by the system.

The nfds parameter is the number of bits to be examined in each of the `fd_set` sets: the function will only check file descriptors `0` through `nfds - 1`, even if some bits are set for descriptors beyond that.

On input, some of the bits of each one of the `fd_set` sets for which the function should wait, should be set using the `FD_SET` macro. `select` examines only those descriptors whose bits are set.

Any of `readfds`, `writefds`, and `exceptfds` can be a `NULL` pointer, if the caller is not interested in testing the corresponding conditions.

On output, if `select` returns a non-negative value, each non-`NULL` argument of the three sets will be replaced with a subset in which a bit is set for every descriptor that was found to be, respectively, ready for input, ready for output, and pending an exceptional condition. Note that if `select` returns -1, meaning a failure, the descriptor sets are *unchanged*, so you should always test the return value before looking at the bits in the returned sets.

The timeout value may be a NULL pointer (no timeout, i.e., wait forever), a pointer to a zero-value structure (poll mode, i.e., test once and exit immediately), or a pointer to a `struct timeval` variable (timeout: `select` will repeatedly test all the descriptors until some of them become ready, or the timeout expires).

`struct timeval` is defined as follows:

```
struct timeval {
time_t tv_sec;
long tv_usec;
};
```

Return Value

On successfull return, `select` returns the number of files ready, or 0, if the timeout expired. The input sets are replaced with subsets that describe which files are ready for which operations. If `select` returns 0 (i.e., the timeout has expired), all the non-`NULL` sets have all their bits reset to zero.

On failure, `select` returns -1, sets `errno` to a suitable value, and leaves the descriptor sets unchanged.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct timeval timeout;
fd_set read_fds, write_fds;
int i, select_result;

timeout.tv_sec = 5; /* 5-second timeout */
timeout.tv_usec = 0;

/* Display status of the 5 files open by default. */
for (i = 0; i < 5; i++)
{

FD_ZERO (&read_fds);
FD_SET (i, &read_fds);
select_result = select (i + 1, &read_fds, 0, 0, &timeout);
if (select_result == -1)
{
fprintf(stderr, "%d: Failure for input", i);
perror("");
}
else
fprintf(stderr,
"%d: %s ready for input\n", i,
select_result ? "" : "NOT");
FD_ZERO (&write_fds);
FD_SET (i, &write_fds);
select_result = select (i + 1, 0, &write_fds, 0, &timeout);
if (select_result == -1)
{
fprintf(stderr, "%d: Failure for output", i);
perror("");
}
else
fprintf(stderr,
"%d: %s ready for output\n", i,
select_result ? "" : "NOT");
}
```

Implementation Notes

The following notes describe some details pertinent to the DJGPP implementation of `select`:

- While `select` waits for the timeout to expire, it repeatedly calls the `__dpmi_yield` function (See __dpmi_yield), so that any other programs that run at the same time (e.g., on Windows) get more CPU time.

- A file handle that belongs to a `FILE` object created by `fopen` or `fdopen` (See fopen) for which `feof` or `ferror` return non-zero, will be reported in the `exceptfds` set; also, such a handle will be reported not input-ready if there are no pending buffered characters in the `FILE` object. This might be a feature or a bug, depending on your point of view; in particular, Unix implementations usually don't check buffered input. Portable programs should refrain from mixing `select` with buffered I/O.

- DOS doesn't support exceptional conditions, so file handles used for unbuffered I/O will *never* be marked in `exceptfds`.

- DOS always returns an output-ready indication for a file descriptor connected to a disk file. So use of `writefds` is only meaningful for character devices.

- The usual text-mode input from the keyboard and other character devices is line-buffered by DOS. This means that if you type one character, `select` will indicate that file handle 0 is ready for input, but a call to `getc` will still block until the **Enter** key is pressed. If you need to make sure that reading a single character won't block, you should read either with BIOS functions such as `getkey` (See getkey) or with raw input DOS functions such as `getch` (See getch), or switch the handle to binary mode with a call to `setmode` (See setmode).

__set_fd_flags

Syntax

```
#include <libc/fd_props.h>

void __set_fd_flags(int fd, unsigned long flags);
```

Description

This internal function adds the combination of flags flags to the flags associated with the file descriptor fd. The flags are some properties that may be associated with a file descriptor (See __set_fd_properties).

The caller should first check that fd has properties associated with it, by calling `__has_fd_properties` (See __has_fd_properties).

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

__set_fd_properties

Syntax

```
#include <libc/fd_props.h>

int __set_fd_properties(int fd, const char *filename, int open_flags);
```

Description

This is an internal function that stores information about the file descriptor fd in a `fd_properties` struct. It is called by `open` and its helper functions.

The file name stored in `fd_properties` is the result of the `_truename` function (See _truename) on filename. The open_flags are scanned and the temporary and append flags are stored in the `flags` field in `fd_properties`.

```
struct fd_properties
{

unsigned char ref_count;
char *filename;
unsigned long flags;
fd_properties *prev;
fd_properties *next;
};
```

`flags` can contain a combination of bits:

`FILE_DESC_TEMPORARY`
: Delete `filename` when `ref_count` becomes zero.

`FILE_DESC_ZERO_FILL_EOF_GAP`
: Tell `write` and `_write` to test for file offset greater than EOF. Set by `lseek` and `llseek`.

`FILE_DESC_DONT_FILL_EOF_GAP`
: Don't test for the EOF gap. Set automatically for stdin, stdout, and NUL. Can also be set by an FSEXT.

`FILE_DESC_PIPE`
: The file descriptor is used in emulating a pipe.

`FILE_DESC_APPEND`
: The file pointer will be set to the end of file before each write.

`FILE_DESC_DIRECTORY`
: The file descriptor is for a directory.

The `flags` can be manipulated using `__set_fd_flags` (See __set_fd_flags), `__clear_fd_flags` (See __clear_fd_flags) and `__get_fd_flags` (See __get_fd_flags).

The file name can be retrieved using `__get_fd_name` (See __get_fd_name).

For more information, see `__clear_fd_properties` (See __clear_fd_properties) and `__dup_fd_properties`

(See __dup_fd_properties).

Return Value

Returns 0 on success. Returns -1 when unable to store the information.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_set_screen_lines

Syntax

```
#include <conio.h>

void _set_screen_lines(int nlines);
```

Description

This function sets the text screen width to 80 and its height to the value given by nlines, which can be one of the following: 25, 28, 35, 40, 43 or 50. On a CGA, only 25-line screen is supported. On an EGA, you can use 25, 35 and 43. VGA, PGA and MCGA support all of the possible dimensions. The number of columns (i.e., screen width) is 80 for all of the above resolutions, because the standard EGA/VGA has no way of changing it. After this function returns, calls to `gettextinfo()` will return the actual screen dimensions as set by `_set_screen_lines()`. That is, you can e.g. test whether `_set_screen_lines()` succeeded by checking the screen height returned by `gettextinfo()` against the desired height. This function has a side effect of erasing the screen contents, so application programs which use it should make their own arrangements to redisplay it.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

setbuf

Syntax

```
#include <stdio.h>

void setbuf(FILE *file, char *buffer);
```

Description

This function modifies the buffering characteristics of file. First, if the file already has a buffer, it is freed. If there was any pending data in it, it is lost, so this function should only be used immediately after a call to `fopen`.

If the buffer passed is `NULL`, the file is set to unbuffered. If a non-`NULL` buffer is passed, it must be at least `BUFSIZ` bytes in size, and the file is set to fully buffered.

See setbuffer. See setlinebuf. See setvbuf.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
setbuf(stdout, malloc(BUFSIZ));
```

setbuffer

Syntax

```
#include <stdio.h>

void setbuffer(FILE *file, char *buffer, int length);
```

Description

This function modifies the buffering characteristics of file. First, if the file already has a buffer, it is freed. If there was any pending data in it, it is lost, so this function should only be used immediately after a call to `fopen`.

If the buffer passed is `NULL`, the file is set to unbuffered. If a non-`NULL` buffer is passed, it must be at least size bytes in size, and the file is set to fully buffered.

See setbuf. See setlinebuf. See setvbuf.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
setbuffer(stdout, malloc(10000), 10000);
```

setcbrk

Syntax

```
#include <dos.h>

void setcbrk(int check);
```

Description

Set the setting of the Ctrl-Break checking flag in MS-DOS. If check is zero, checking is not done. If nonzero, checking is done.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

_setcursortype

Syntax

```
#include <conio.h>

void _setcursortype(int _type);
```

Description

Sets the cursor type. _type is one of the following:

`_NOCURSOR`
: No cursor is displayed.

`_SOLIDCURSOR`

A solid block is displayed.

`_NORMALCURSOR`
An underline cursor is displayed.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

setdate

Syntax

```
#include <dos.h>

void setdate(struct date *ptr);
```

Description

This function sets the current time.

For the description of `struct date`, see See getdate. Also see See settime.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct date d;
setdate(&d);
```

setdisk

Syntax

```
#include <dir.h>

int setdisk(int drive);
```

Description

This function sets the current disk (0=A).

See also See getdisk.

Return Value

The highest drive actually present that the system can reference.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("There are %d drives\n", setdisk(getdisk()));
```

setenv

Syntax

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int rewrite);
```

Description

This function sets the environment variable name to value. If rewrite is set, then this function will replace any existing value. If it is not set, it will only put the variable into the environment if that variable isn't already defined.

Return Value

Zero on success, -1 on failure; `errno` is set to the reason for failure: `EINVAL` if the name parameter is `NULL` or points to an empty string, or `ENOMEM` if there was insufficient memory to add the variable to the environment.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. This function is new to the Posix 1003.1-200x draft. Our implementation allows name to contain a `=`, while the POSIX spec does not; the portion of name up to but not including the `=` will be used as the name for the environment variable.

setftime

Syntax

```
#include <dos.h>

int setftime(int handle, struct ftime *ftimep);
```

Description

This function sets the modification time of a file. Note that since writing to a file and closing a file opened for writing also sets the modification time, you should only use this function on files opened for reading.

See getftime, for the description of `struct ftime`.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int q = open("data.txt", O_RDONLY);
struct ftime f;
f.ft_tsec = f.ft_min = f.ft_hour = f.ft_day = f.ft_month = f.ft_year = 0;
setftime(q, &f);
close(q);
```

setgid

Syntax

```
#include <unistd.h>

int setgid(gid_t gid);
```

Description

This function is simulated, since MS-DOS does not support group IDs.

Return Value

If gid is equal to that returned by See getgid, returns zero. Otherwise, returns -1 and sets `errno` to EPERM.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

setgrent

Syntax

```
#include <grp.h>

void setgrent(void);
```

Description

This function should be called before any call to `getgrent`, `getgrgid`, or `getgrnam`, to start searching the groups' list from the beginning. See getgrent.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

setitimer

Syntax

```
#include <sys/time.h>

extern long __djgpp_clock_tick_interval;

struct timeval {
time_t tv_sec;
long tv_usec;
};

struct itimerval {
struct timeval it_interval; /* timer interval */
struct timeval it_value; /* current value */
};

int setitimer(int which, struct itimerval *value,
struct itimerval *ovalue);
```

Description

Each process has two interval timers, `ITIMER_REAL` and `ITIMER_PROF`, which raise the signals `SIGALRM` and `SIGPROF`, respectively. These are typically used to provide `alarm` and profiling capabilities.

This function changes the current value of the interval timer specified by which to the values in structure value. The previous value of the timer is returned in ovalue if it is not a `NULL` pointer. When the timer expires, the appropriate signal is raised.

Please see the documentation for `signal` (See signal) for restrictions on signal handlers.

If value is a `NULL` pointer, `setitimer` stores the previous timer value in ovalue (if it is non-`NULL`), like `getitimer` does, but otherwise does nothing.

A timer is defined by the `itimerval` structure. If the `it_value` member is non-zero it specifies the time to the next timer expiration. If `it_interval` is non-zero, it specifies the value with which to reload the timer upon expiration. Setting `it_value` to zero disables a timer. Setting `it_interval` to zero causes the timer to stop after the next expiration (assuming that `it_value` is non-zero).

Although times can be given with microsecond resolution, the granularity is determined by the timer interrupt frequency. Time values smaller than the system clock granularity will be rounded up to that granularity, before they are used. This means that passing a very small but non-zero value in `value->it_interval.tv_usec` will cause the system clock granularity to be stored and returned by the next call to `getitimer`. See the example below.

If an application changes the system clock speed by reprogramming the timer chip, it should make the new clock speed known to `setitimer`, otherwise intervals smaller than the default PC clock speed cannot be set with a call to `setitimer` due to rounding up to clock granularity. To this end, an external variable `__djgpp_clock_tick_interval` is provided, which should be set to the number of microseconds between two timer ticks that trigger Interrupt 8. The default value of this variable is `-1`, which causes `setitimer` to work with 54926 microsecond granularity that corresponds to the standard 18.2Hz clock frequency. The library never changes the value of `__djgpp_clock_tick_interval`.

Return Value

Returns 0 on success, -1 on failure (and sets `errno`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Bugs

This version uses `uclock` (See uclock) to determine the time of expiration. Under Windows 3.X, this fails because the OS reprograms the timer. Under Windows 9X, `uclock` sometimes reports erratic (non-increasing) time values; in these cases the timer might fire at a wrong time.

A misfeature of Windows 9X prevents the timer tick interrupt from being delivered to programs that are in the background (i.e. don't have the focus), even though the program itself might continue to run, if you uncheck the Background: Always suspend property in the Property Sheets. Therefore, the timers will not work in background programs on Windows 9X.

Also, debuggers compiled with DJGPP v2.02 and earlier cannot cope with timers and report `SIGSEGV` or `SIGABRT`, since signals were not supported in a debugged program before DJGPP v2.03.

Example

```
/* Find out what is the system clock granularity. */

struct itimerval tv;

tv.it_interval.tv_sec = 0;
tv.it_interval.tv_usec = 1;
tv.it_value.tv_sec = 0;
tv.it_value.tv_usec = 0;
setitimer (ITIMER_REAL, &tv, 0);
setitimer (ITIMER_REAL, 0, &tv);
printf ("System clock granularity: %ld microseconds.\n",
tv.it_interval.tv_usec);
```

setjmp

Syntax

```
#include <setjmp.h>

int setjmp(jmp_buf j);
```

Description

This function stores the complete CPU state into j. This information is complete enough that `longjmp` (See longjmp) can return the program to that state. It is also complete enough to implement coroutines.

Return Value

This function will return zero if it is returning from its own call. If longjmp is used to restore the state, it will return whatever value was passed to longjmp, except if zero is passed to longjmp it will return one.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
jmp_buf j;
if (setjmp(j))
return;
do_something();
longjmp(j, 1);
```

setlinebuf

Syntax

```
#include <stdio.h>

void setlinebuf(FILE *file);
```

Description

This function modifies the buffering characteristics of file. First, if the file already has a buffer, it is freed. If there was any pending data in it, it is lost, so this function should only be used immediately after a call to `fopen`.

Next, a buffer is allocated and the file is set to line buffering.

See setbuf. See setlinebuf. See setvbuf.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
setlinebuf(stderr);
```

setlocale

Syntax

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

Description

This function sets part or all of the current locale. The category is one of the following:

`LC_ALL`
: Set all parts of the locale.

`LC_COLLATE`
: Set the collating information.

`LC_CTYPE`
: Set the character type information.

`LC_MONETARY`
: Set the monetary formatting information.

`LC_NUMERIC`
: Set the numeric formatting information.

`LC_TIME`
: Set the time formatting information.

The locale should be the name of the current locale. Currently, only the "C" and "POSIX" locales are supported. If the locale is NULL, no action is performed. If locale is "", the locale is identified by environment variables (currently not supported).

See localeconv.

Return Value

A static string naming the current locale for the given category, or NULL if the requested locale is not supported.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
setlocale(LC_ALL, "C");
```

setmntent

Syntax

```
#include <mntent.h>

FILE *setmntent(char *filename, const char *mode);
```

Description

This function returns an open `FILE*` pointer which can be used by `getmntent` (See getmntent). The arguments filename and mode are always ignored under MS-DOS, but for portability should be set, accordingly, to the name of the file which describes the mounted filesystems and the open mode of that file (like the mode argument to `fopen`, See fopen). (There is no single standard for the name of the file that keeps the mounted filesystems, but it is usually, although not always, listed in the header `<mntent.h>`.)

Return Value

The `FILE*` pointer is returned. For MS-DOS, this `FILE*` is not a real pointer and may only be used by `getmntent`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <mntent.h>
#if defined(MNT_MNTTAB)
#define MNTTAB_FILE MNT_MNTTAB
#elif defined(MNTTABNAME)
#define MNTTAB_FILE MNTTABNAME
#else
#define MNTTAB_FILE "/etc/mnttab"
#endif

FILE *mnt_fp = setmntent (MNTTAB_FILE, "r");
```

setmode

Syntax

```
#include <io.h>

int setmode(int file, int mode);
```

Description

This function sets the mode of the given file to mode, which is either `O_TEXT` or `O_BINARY`. It will also set the file into either cooked or raw mode accordingly, and set any `FILE*` objects that use this file into text or binary mode.

When called to put file that refers to the console into binary mode, `setmode` will disable the generation of the signals `SIGINT` and `SIGQUIT` when you press, respectively, **Ctrl-C** and **Ctrl-** (**Ctrl-BREAK** will still cause `SIGINT`), because many programs that use binary reads from the console will also want to get the `^C` and `^\` keys. You can use the `__djgpp_set_ctrl_c` library function (See __djgpp_set_ctrl_c) if you want **Ctrl-C** and **Ctrl-** to generate signals while console is read in binary mode.

Note that, for buffered streams (`FILE*`), you must call `fflush` (See fflush) before `setmode`, or call `setmode` before writing anything to the file, for proper operation.

Return Value

When successful, the function will return the previous mode of the given file. In case of failure, -1 is returned and `errno` is set.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
setmode(0, O_BINARY);
```

setpgid

Syntax

```
#include <unistd.h>

int setpgid(pid_t _pid, pid_t _pgid);
```

Return Value

-1 (EPERM) if _pgid is not your current pid, else zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

setpwent

Syntax

```
#include <pwd.h>

void setpwent(void);
```

Description

This function reinitializes `getpwent` so that scanning will start from the start of the list. See getpwent.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

setrlimit

Syntax

```
#include <sys/resource.h>

int setrlimit (int rltype, const struct rlimit *rlimitp);
```

Description

This function sets new limit pointed to by rlimitp on the resource limit specified by rltype. Note that currently it always fails.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

setsid

Syntax

```
#include <unistd.h>

pid_t setsid(void);
```

Description

This function does not do anything. It exists to assist porting from Unix.

Return Value

Return value of See getpid.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

setstate

Syntax

```
#include <stdlib.h>

char *setstate(char *arg_state);
```

Description

Restores the random number generator (See random) state from pointer arg_state to state array.

Return Value

Pointer to old state information.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

settime

Syntax

```
#include <dos.h>

void settime(struct time *ptr);
```

Description

This function sets the current time.

For the description of `struct time`, see See gettime. Also see See setdate.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct time t;
settime(&t);
```

settimeofday

Syntax

```
#include <time.h>

int settimeofday(struct timeval *tp, ...);
```

Description

Sets the current GMT time. For compatibility, a second argument is accepted. See gettimeofday, for information on the structure types.

Return Value

Zero if the time was set, nonzero on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

setuid

Syntax

```
#include <unistd.h>

int setuid(uid_t uid);
```

Description

This function is simulated, since MS-DOS does not support user IDs.

Return Value

If uid is equal to that returned by See getuid, returns zero. Otherwise, returns -1 and sets `errno` to EPERM.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

setvbuf

Syntax

```
#include <stdio.h>

int setvbuf(FILE *file, char *buffer, int type, int length);
```

Description

This function modifies the buffering characteristics of file. First, if the file already has a buffer, it is freed. If there was any pending data in it, it is lost, so this function should only be used immediately after a call to `fopen`.

If the type is `_IONBF`, the buffer and length are ignored and the file is set to unbuffered mode.

If the type is `_IOLBF` or `_IOFBF`, then the file is set to line or fully buffered, respectively. If buffer is `NULL`, a buffer of size size is created and used as the buffer. If buffer is non-`NULL`, it must point to a buffer of at least size size and will be used as the buffer.

See setbuf. See setbuffer. See setlinebuf.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
setvbuf(stderr, NULL, _IOLBF, 1000);
```

sigaction

Syntax

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *act,
struct sigaction *oact);
```

Description

This function allows to examine and/or change the action associated with a signal sig. The `struct sigaction` structure, defined by the header file `signal.h`, is declared as follows:

```
struct sigaction {
int sa_flags; /* flags for the action;
* currently ignored */
void (*sa_handler)(int); /* the handler for the signal */
sigset_t sa_mask; /* additional signals to be blocked */
};
```

The `sa_handler` member is a signal handler, see See signal. The `sa_mask` member defines the signals, in addition to sig, which are to be blocked during the execution of `sa_handler`.

The `sigaction` function sets the structure pointed to by `oact` to the current action for the signal sig, and then sets the new action for sig as specified by act. If the act argument is `NULL`, `sigaction` returns the current signal action in oact, but doesn't change it. If the oact argument is a `NULL` pointer, it is ignored. Thus, passing `NULL` pointers for both `act` and `oact` is a way to see if sig is a valid signal number on this system (if not, `sigaction` will return -1 and set `errno`).

Return Value

0 on success, -1 for illegal value of sig.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

sigaddset

Syntax

```
#include <signal.h>

int sigaddset (sigset_t *set, int signo)
```

Description

This function adds the individual signal specified by signo the set of signals pointed to by set.

Return Value

0 upon success, -1 if set is a NULL pointer, or if signo is specifies an unknown signal.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

sigdelset

Syntax

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

```
int sigdelset (sigset_t *set, int signo)
```

Description

This function removess the individual signal specified by signo from the set of signals pointed to by set.

Return Value

0 upon success, -1 if set is a NULL pointer, or if signo is specifies an unknown signal.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

sigemptyset

Syntax

```
#include <signal.h>

int sigemptyset (sigset_t *set)
```

Description

This function initializes the set of signals pointed to by set to exclude all signals known to the DJGPP runtime system. Such an empty set, if passed to `sigprocmask` (See sigprocmask), will cause all signals to be passed immediately to their handlers.

Return Value

0 upon success, -1 if set is a NULL pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

sigfillset

Syntax

```
#include <signal.h>

int sigfillset (sigset_t *set)
```

Description

This function initializes the set of signals pointed to by set to include all signals known to the DJGPP runtime system. Such a full set, if set by `sigprocmask` (See sigprocmask), will cause all signals to be blocked from delivery to their handlers. Note that the set returned by this function only includes signals in the range `SIGABRT..SIGTRAP`; software interrupts and/or user-defined signals aren't included.

Return Value

0 upon success, -1 if set is a NULL pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
sigset_t full_set, prev_set;

sigfillset (&full_set);
sigprocmask (SIG_UNBLOCK, &full_set, &prev_set);
```

sigismember

Syntax

```
#include <signal.h>

int sigismember (sigset_t *set, int signo)
```

Description

This function checks whether the signal specified by signo is a member of the set of signals pointed to by set.

Return Value

1 if the specified signal is a member of the set, 0 if it isn't, or if signo specifies an unknown signal, -1 if set is a NULL pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

siglongjmp

Syntax

```
#include <setjmp.h>

int siglongjmp(sigjmp_buf env, int val);
```

Description

See longjmp.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

signal

Syntax

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Description

Signals are generated in response to some exceptional behavior of the program, such as division by 0. A signal can also report some asynchronous event outside the program, such as someone pressing a **Ctrl-BREAK** key combination.

Signals are numbered 0..255 for software interrupts and 256..287 for exceptions (exception number plus 256); other implementation-specific codes are specified in `<signal.h>` (see below). Every signal is given a mnemonic which you should use for portable programs.

By default, signal `SIGQUIT` is discarded. This is so programs ported from other DOS environments, where `SIGQUIT` is generally not supported, continue to work as they originally did. If you want `SIGQUIT` to abort with a traceback, install `__djgpp_traceback_exit` as its handler (See __djgpp_traceback_exit).

The default handling for the rest of the signals is to print a traceback (a stack dump which describes the sequence of function calls leading to the generation of the signal) and abort the program by calling `_exit` (See _exit). As an exception, the default handler for the signal `SIGINT` doesn't print the traceback, and calls `exit` instead of `_exit`, when the INTR key (**Ctrl-C** by default) is pressed, so that programs could be shut down safely in this manner. `SIGINT` raised by **Ctrl-BREAK** does generate the traceback.

The function `signal` allows you to change the default behavior for a specific signal. It registers func as a signal handler for signal number sig. After you register your function as the handler for a particular signal, it will be called when that signal occurs. The execution of the program will be suspended until the handler returns or calls `longjmp` (See longjmp).

The state of the floating-point unit (FPU) is not saved, before entering a signal handler. It can be extremely costly to save the FPU state. Most signal handlers do not use floating-point operations, so the overhead of saving FPU state is avoided. An example of a signal handler that saves the FPU state is the function `dbgsig` in `src/debug/common/dbgcom.c` in the DJGPP sources.

You may pass `SIG_DFL` as the value of func to reset the signal handling for the signal sig to default (also See __djgpp_exception_toggle, for a quick way to restore all the signals' handling to default), `SIG_ERR` to force an error when that signal happens, or `SIG_IGN` to ignore that signal. Signal handlers that you write are regular C functions, and may call any function that the ANSI/POSIX specs say are valid for signal handlers. For maximum portability, a handler for hardware interrupts and processor exceptions should only make calls to `signal`, assign values to data objects of type `volatile sig_atomic_t` (defined as `int` on `<signal.h>`) and return. Handlers for hardware interrupts need also be locked in memory (so that the operation of virtual memory mechanism won't swap them out), See __dpmi_lock_linear_region, locking memory regions. Handlers for software interrupts can also terminate by calling `abort`, `exit` or `longjmp`.

The following signals are defined on `<signal.h>`:

`SIGABRT`
: The Abort signal. Currently only used by the `assert` macro to terminate the program when an assertion fails (See assert), and by the `abort` function (See abort).

`SIGFPE`
: The Floating Point Error signal. Generated in case of divide by zero exception (Int 00h), overflow exception (Int 04h), and any x87 co-processor exception, either generated by the CPU (Int 10h), or by the co-processor itself (Int 75h). The co-processor status word is printed by the default handler for this signal. See _status87, for the definition of the individual bits of the status word.

: The DJGPP startup code masks all numeric exceptions, so this signal is usually only triggered by an integer divide by zero operation. If you want to unmask some of the numeric exceptions, see See _control87.

`SIGILL`
: The Invalid Execution signal. Currently only generated for unknown/invalid exceptions.

`SIGINT`
: The Interrupt signal. Generated when an INTR key (**Ctrl-C** by default) or **Ctrl-BREAK** (Int 1Bh) key is hit. Note that when you open the console in binary mode, or switch it to binary mode by a call to `setmode` (See setmode), generation of `SIGINT` as result of **Ctrl-C** key is disabled. This is so for programs (such as Emacs) which want to be able to read the `^C` character as any other character. Use the library function `__djgpp_set_ctrl_c` to restore `SIGINT` generation when **Ctrl-C** is hit, if you need this. See __djgpp_set_ctrl_c, for details on how this should be done. **Ctrl-BREAK** always generates `SIGINT`.

: DJGPP hooks the keyboard hardware interrupt (Int 09h) to be able to generate `SIGINT` in response to the INTR key; you should be aware of this when you install a handler for the keyboard interrupt.

: Note that the key which generates `SIGINT` can be changed with a call to `__djgpp_set_sigint_key` function. See __djgpp_set_sigint_key.

`SIGSEGV`
: The invalid storage access (Segmentation Violation) signal. Generated in response to any of the following exceptions: Bound range exceeded in BOUND instruction (Int 05h), Double Exception or an exception in the exception handler (Int 08h), Segment Boundary violation by co-processor (Int 09h), Invalid TSS (Int 0Ah), Segment Not Present (Int 0Bh), Stack Fault (Int 0Ch), General Protection Violation (Int 0Dh), or Page Fault (Int 0Eh). Note that Int 09h is only generated on 80386 processor; i486 and later CPUs cause Int 0Dh when the co-processor accesses memory out of bounds. The Double Exception, Invalid TSS, Segment Not Present, Stack Fault, GPF, and Page Fault exceptions will cause an error code to be printed, if it is non-zero.

`SIGTERM`
: The Termination Request signal. Currently unused.

: The signals below this are not defined by ANSI C, and cannot be used when compiling under `-ansi` option to `gcc`.

`SIGALRM`
: The Alarm signal. Generated after certain time period has passed after a call to `alarm` library function (See alarm).

`SIGHUP`
: The Hang-up signal. Currently unused.

`SIGKILL`
: The Kill signal. Currently unused.

`SIGPIPE`

The Broken Pipe signal. Currently unused.

`SIGQUIT`
The Quit signal. Generated when the QUIT key (**Ctrl-** by default) is hit. The key that raises the signal can be changed with a call to `__djgpp_set_sigquit_key` function. See __djgpp_set_sigquit_key. By default, `SIGQUIT` is discarded, even if its handler is `SIG_DFL`, so that DOS programs which don't expect it do not break. You can change the effect of `SIGQUIT` to abort with traceback by installing `__djgpp_traceback_exit` as its handler. See __djgpp_traceback_exit.

DJGPP hooks the keyboard hardware interrupt (Int 09h) to be able to generate `SIGQUIT` in response to the QUIT key; you should be aware of this when you install a handler for the keyboard interrupt.

`SIGUSR1`
User-defined signal no. 1.

`SIGUSR2`
User-defined signal no. 2.

The signals below are not defined by ANSI C and POSIX, and cannot be used when compiling under either `-ansi` or `-posix` options to `gcc`.

`SIGTRAP`
The Trap Instruction signal. Generated in response to the Debugger Exception (Int 01h) or Breakpoint Exception (Int 03h).

`SIGNOFP`
The No Co-processor signal. Generated if a co-processor (floating-point) instruction is encountered when no co-processor is installed (Int 07h).

`SIGTIMR`
The Timer signal. Used by the `setitimer` and `alarm` functions (See setitimer, and See alarm).

`SIGPROF`
The Profiler signal. Used by the execution profile gathering code in a program compiled with `-pg` option to `gcc`.

Return Value

The previous handler for signal sig, or `SIG_ERR` if the value of sig is outside legal limits.

Signal Mechanism Implementation Notes

Due to subtle aspects of protected-mode programs operation under MS-DOS, signal handlers cannot be safely called from hardware interrupt handlers. Therefore, DJGPP exception-handling mechanism arranges for the signal handler to be called on the first occasion that the program is in protected mode and touches any of its data. This means that if the exception occurs while the processor is in real mode, like when your program calls some DOS service, the signal handler won't be called until that call returns. For instance, if you call `read` (or `scanf`, or `gets`) to read text from the console and press **Ctrl-C**, you will have to press **Enter** to terminate the `read` call to cause the signal handler for `SIGINT` to be called. Another significant implication of this implementation is that when the program isn't touching any of its data (like in very tight loops which only use values in the registers), it cannot be interrupted.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

sigpending

Syntax

```
#include <signal.h>

int sigpending (sigset_t *set)
```

Description

This function retrieves the signals that have been sent to the program, but are being blocked from delivery by the program's signal mask (See sigprocmask). The bit-mapped value which describes the pending signals is stored in the structure pointed to by set. You can use the `sigismember` function (See sigismember) to see what individual signals are pending.

Return Value

0 on success, -1 on failure (and errno set to `EFAULT`).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
#include <signal.h>

sigset_t pending_signals;

/* If SIGINT is pending, force it to be raised. */
if (sigpending (&pending_signals) == 0
&& sigismember (&pending_signals, SIGINT))
{
sigset_t new_set, old_set;
sigemptyset (&new_set);
sigaddset (&new_set, SIGINT);

/* This sigprocmask() call will raise SIGINT. */
sigprocmask (SIG_UNBLOCK, &new_set, &old_set);

/* Restore mask */
sigprocmask (SIG_SETMASK, &old_set, &new_set);
}
```

sigprocmask

Syntax

```
#include <signal.h>

int sigprocmask (int how, const sigset_t *new_set, sigset_t *old_set)
```

Description

This function is used to examine and/or change the program's current signal mask. The current signal mask determines which signals are blocked from being delivered to the program. A signal is blocked if its bit in the mask is set. (See sigismember, See sigaddset, See sigdelset, See sigemptyset, See sigfillset, for information about functions to manipulate the signal masks.) When a blocked signal happens, it is not delivered to the program until such time as that signal is unblocked by another call to `sigprocmask`. Thus blocking a signal is an alternative to ignoring it (by setting its handler to `SIG_IGN`, See signal), but has an advantage of not missing the signal entirely.

The value of the argument how determines the operation: if it is `SIG_BLOCK`, the set pointed to by the argument new_set is *added* to the current signal mask. If the value is `SIG_UNBLOCK`, the set pointed to by new_set is *removed* from the current signal mask. If the value is `SIG_SETMASK`, the current mask is *replaced* by the set pointed to by new_set.

If the argument old_set is not `NULL`, the previous mask is stored in the space pointed to by old_set. If the value of the argument new_set is `NULL`, the value of how is not significant and the process signal mask is unchanged; thus, the call with a zero new_set can be used to inquire about currently blocked signals, without changing the current set.

If the new set defined by the call causes some pending signals to be unblocked, they are all delivered (by calling `raise`) before the call to `sigprocmask` returns.

The DJGPP implementation only records a single occurrence of any given signal, so when the signal is unblocked, its handler will be called at most once.

It is not possible to block CPU exceptions such as Page Fault, General Protection Fault etc. (mapped to `SIGSEGV` signal); for these, `sigprocmask` will behave as if the call succeeded, but when an exception happens, the signal handler will be called anyway (the default handler will abort the program).

Also note that there are no provisions to save and restore any additional info about the signal beyond the fact that it happened. A signal handler might need such info to handle the signal intelligently. For example, a handler for `SIGFPE` might need to examine the status word of the FPU to see what exactly went wrong. But if the signal was blocked and is delivered after a call to `sigprocmask` has unblocked it, that information is lost. Therefore, if you need access to such auxiliary information in the signal handler, don't block that signal.

Return Value

0 on success, -1 for illegal value of sig or illegal address in new_set or old_set.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

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

static void
sig_catcher (int signo)
{

cprintf ("\r\nGot signal %d\r\n", signo);
}


int
main (void)
{

sigset_t sigmask, prevmask;

signal (SIGINT, sig_catcher);

sigemptyset (&sigmask);

sigaddset (&sigmask, SIGINT);
if (sigprocmask (SIG_SETMASK, &sigmask, &prevmask) == 0)
cputs ("SIGINT blocked. Try to interrupt me now.\r\n");
while (!kbhit ())
;
cputs ("See? I wasn't interrupted.\r\n");
cputs ("But now I will unblock SIGINT, and then get the signal.\r\n");
sigprocmask (SIG_UNBLOCK, &sigmask, &prevmask);
return 0;
}
```

sigsetjmp

Syntax

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

Description

See setjmp.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

sin

Syntax

```
#include <math.h>

double sin(double x);
```

Description

This function computes the sine of x (which should be given in radians).

Return Value

The sine of x. If the absolute value of x is finite but greater than or equal to 2^63, the value is 0 (since for arguments that large each bit of the mantissa is more than `Pi`). If the value of x is infinite or `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Accuracy

In general, this function's relative accuracy is about 1.7*10^(-16), which is close to the machine precision for a `double`. However, for arguments very close to `Pi` and its odd multiples, the relative accuracy can be many times worse, due to loss of precision in the internal FPU computations. Since sin(Pi) is zero, the absolute accuracy is still very good; but if your program needs to preserve high *relative* accuracy for such arguments, link with `-lm` and use the version of `sin` from `libm.a` which does elaborate argument reduction, but is about three times slower.

sincos

Syntax

```
#include <math.h>

void sincos(double *cosine, double *sine, double x);
```

Description

This function computes the cosine and the sine of x in a single call, and stores the results in the addresses pointed to by cosine and sine, respectively. Since the function exploits a machine instruction that computes both cosine and sine simultaneously, it is faster to call `sincos` than to call `cos` and `sin` for the same argument.

If the absolute value of x is finite but greater than or equal to 2^63, the value stored in *cosine is 1 and the value stored in *sine is 0 (since for arguments that large each bit of the mantissa is more than `Pi`). If the value of x is infinite or `NaN`, `NaN` is stored in both *cosine and *sine, and `errno` is set to `EDOM`.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

sinh

Syntax

```
#include <math.h>

double sinh(double x);
```

Description

This function computes the hyperbolic sine of x.

Return Value

The hyperbolic sine of x. If the absolute value of x is finite but so large that the result would overflow a `double`, the return value is `Inf` with the same sign as x, and `errno` is set to `ERANGE`. If x is either a positive or a negative infinity, the result is `+Inf` with the same sign as x, and `errno` is not changed. If x is `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

sleep

Syntax

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```

Description

This function causes the program to pause for seconds seconds.

Return Value

The number of seconds that haven't passed (i.e. always zero)

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
sleep(5);
```

snprintf

Syntax

```
#include <stdio.h>

int snprintf (char *buffer, size_t n, const char *format,
...);
```

Description

This function works similarly to `sprintf()` (See sprintf), but the size n of the buffer is also taken into account. This function will write n - 1 characters. The nth character is used for the terminating nul. If n is zero, buffer is not touched.

Return Value

The number of characters that would have been written (excluding the trailing nul) is returned; otherwise -1 is returned to flag encoding or buffer space errors.

The maximum accepted value of n is `INT_MAX`. `INT_MAX` is defined in `<limits.h>`. -1 is returned and `errno` is set to `EFBIG`, if n is greater than this limit.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1)

Notes:

1. The buffer size limit is imposed by DJGPP. Other systems may not have this limitation.

__solve_dir_symlinks

Syntax

```
#include <libc/symlink.h>

int __solve_dir_symlinks(const char *symlink_path, char *real_path);
```

Description

This function resolves given symlink in symlink_path---all path components **except** the last one and all symlink levels are resolved. If symlink_path does not contain symlinks at all, it is simply copied to real_path.

real_path should be of size `FILENAME_MAX`, to contain the maximum possible length of path.

Return Value

Zero in case of error (and `errno` set to the appropriate error code), non-zero in case of success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <libc/symlink.h>
#include <stdio.h>

char fn[] = "c:/somelink/someotherlink/somefile";
char file_name[FILENAME_MAX];

__solve_dir_symlinks(fn, file_name);
printf("The real path to %s is %s\n", fn, file_name);
```

__solve_symlinks

Syntax

```
#include <libc/symlink.h>

int __solve_symlinks(const char *symlink_path, char *real_path);
```

Description

This function fully resolves given symlink in symlink_path---all path components and all symlink levels are resolved. The returned path in real_path is guaranteed to be symlink-clean and understandable by DOS. If symlink_path does not contain symlinks at all, it is simply copied to real_path.

real_path should be of size `FILENAME_MAX`, to contain the maximum possible length of path.

Return Value

Zero in case of error (and `errno` set to the appropriate error code), non-zero in case of success.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <libc/symlink.h>
#include <stdio.h>

const char fn[] = "c:/somedir/somelink";
char file_name[FILENAME_MAX];

__solve_symlinks(fn, file_name);
printf("File %s is really %s\n", fn, file_name);
```

sound

Syntax

```
#include <pc.h>

void sound(int _frequency);
```

Description

Enables the PC speaker at the given frequency. The argument _frequency should be given in Hertz units.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

spawn*

Syntax

```
#include <process.h>

int spawnl(int mode, const char *path, const char *argv0, ..., NULL);
int spawnle(int mode, const char *path, const char *argv0, ...,
NULL /*, const char **envp */);
int spawnlp(int mode, const char *path, const char *argv0, ..., NULL);
int spawnlpe(int mode, const char *path, const char *argv0, ...,
NULL /*, const char **envp */);

int spawnv(int mode, const char *path, char *const argv[]);
int spawnve(int mode, const char *path, char *const argv[],
char *const envp[]);
int spawnvp(int mode, const char *path, char *const argv[]);
int spawnvpe(int mode, const char *path, char *const argv[],
char *const envp[]);
```

Description

These functions run other programs. The path points to the program to run, and may optionally include its extension. These functions will look for a file path with the extensions `.com`, `.exe`, `.bat`, and `.btm`; if none are found, neither in the current directory nor along the `PATH`, they will look for path itself.

`.com` programs are invoked via the usual DOS calls; DJGPP `.exe` programs are invoked in a way that allows long command lines to be passed; other `.exe` programs are invoked via DOS; `.bat` and `.btm` programs are invoked via the command processor given by the `COMSPEC` environment variable; `.sh`, `.ksh` programs and programs with any other extensions that have `#!` as their first two characters are assumed to be Unix-style scripts and are invoked by calling a program whose pathname immediately follows the first two characters. (If the name of that program is a Unix-style pathname, without a drive letter and without an extension, like `/bin/sh`, the `spawn` functions will additionally look them up on the `PATH`; this allows to run Unix scripts without editing, if you have a shell installed somewhere along your `PATH`.) Any non-recognizable files will be also invoked via DOS calls.

WARNING! DOS is rather stupid in invoking programs: if the file doesn't have the telltale ‘‘MZ’’ signature of the `.exe` style programs, DOS assumes it is a `.com` style image and tries to execute it directly. If the file is not really an executable program, your application will almost certainly crash. Applications that need to be robust in such situations should test whether the program file is indeed an executable, e.g. with calls to `stat` (See stat) or `_is_executable` (See _is_executable) library functions.

Note that built-in commands of the shells can *not* be invoked via these functions; use `system` instead, or invoke the appropriate shell with the built-in command as its argument.

The programs are invoked with the arguments given. The zeroth argument is normally not used, since MS-DOS cannot pass it separately, but for compatibility it should be the name of the program. There are two ways of passing arguments. The `l` functions (like `spawnl`) take a list of arguments, with a `NULL` at the end of the list. This is useful when you know how many argument there will be ahead of time. The `v` functions (like `spawnv`) take a pointer to a list of arguments, which also must be `NULL`-terminated. This is useful when you need to compute the number of arguments at runtime.

In either case, you may also specify `e` to indicate that you will be giving an explicit environment, else the current environment is used. You may also specify `p` to indicate that you would like `spawn*` to search the `PATH` (in either the environment you pass or the current environment) for the executable, else it will only check the explicit path given.

Note that these function understand about other DJGPP programs, and will call them directly, so that you can pass command lines longer than 126 characters to them without any special code. DJGPP programs called by these functions will *not* glob the arguments passed to them; other programs also won't glob the arguments if they suppress expansion when given quoted filenames.

When the calling program runs on Windows 9X or Windows 2000 and calls the system shell to run the child program, or if the child program is a native Windows program (in `PE-COFF` format), or when the system shell is

4DOS or NDOS and the shell is called to run the command, command lines longer than 126 characters are passed via the environment variable CMDLINE.

See exec*.

Return Value

If successful and mode is P_WAIT, these functions return the exit code of the child process in the lower 8 bits of the return value. Note that if the program is run by a command processor (e.g., if it's a batch file), the exit code of that command processor will be returned. COMMAND.COM is notorious for returning 0 even if it couldn't run the command.

If successful and mode is P_OVERLAY, these functions will not return.

If there is an error (e.g., the program specified as argv[0] cannot be run, or the command line is too long), these functions return -1 and set errno to indicate the error. If the child program was interrupted by **Ctrl-C** or a Critical Device error, errno is set to EINTR (even if the child's exit code is 0), and bits 8-17 of the return value are set to SIGINT or SIGABRT, accordingly. Note that you must set the signal handler for SIGINT to SIG_IGN, or arrange for the handler to return, or else your program will be aborted before it will get chance to set the value of the return code.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *environ[] = {
"PATH=c:\\dos;c:\\djgpp;c:\\usr\\local\\bin",
"DJGPP=c:/djgpp",
0
};

char *args[] = {
"gcc",
"-v",
"hello.c",
0
};

spawnvpe(P_WAIT, "gcc", args, environ);
```

sprintf

Syntax

```
#include <stdio.h>

int sprintf(char *buffer, const char *format, ...);
```

Description

Sends formatted output from the arguments (…) to the buffer. See printf.

To avoid buffer overruns, it is safer to use snprintf() (See snprintf).

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

sqrt

Syntax

```
#include <math.h>

double sqrt(double x);
```

Description

This function computes the square root of x.

Return Value

The square root of x. If x is negative or a `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

srand

Syntax

```
#include <stdlib.h>

void srand(unsigned seed);
```

Description

Initializes the random number generator for `rand()`. If you pass the same seed, `rand()` will return the same sequence of numbers. You can seed from `time` (See time) or `rawclock` (See rawclock).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
/* random pause */
srand(time(0));
for (i=rand(); i; i--);
```

srandom

Syntax

```
#include <stdlib.h>

int srandom(int seed);
```

Description

Initializes the random number generator (See random). Passing the same seed results in `random` returning predictable sequences of numbers, unless See initstate or See setstate are called.

Return Value

Zero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
srandom(45);
```

sscanf

Syntax

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

```
int sscanf(const char *string, const char *format, ...);
```

Description

This function scans formatted text from the string and stores it in the variables pointed to by the arguments. See scanf.

Return Value

The number of items successfully scanned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Stack overflow handler

Syntax

```
extern unsigned int __djgpp_stack_overflow_eip;
void __djgpp_stack_overflow_exit(void);
```

Description

This is the stack overflow handler, an internal function intended to be used only by special code generated by the compiler. This will exit with a suitable error message.

Return Value

This function does not return.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

stackavail

Syntax

```
#include <stdlib.h>

int stackavail(void);
```

Description

This function returns the number of bytes that are available on the stack.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("Available stack size is %d bytes\n", stackavail());
```

stat

Syntax

```
#include <sys/stat.h>

int stat(const char *file, struct stat *sbuf);
```

Description

This function obtains the status of the file file and stores it in sbuf. `stat` follows symbolic links. To get information about a symbolic link, use `lstat` (See lstat) instead.

sbuf has this structure:

```
struct stat {
```

```
time_t st_atime; /* time of last access */
time_t st_ctime; /* time of file's creation */
dev_t st_dev; /* The drive number (0 = a:) */
gid_t st_gid; /* what getgid() returns */
ino_t st_ino; /* starting cluster or unique identifier */
mode_t st_mode; /* file mode - S_IF* and S_IRUSR/S_IWUSR */
time_t st_mtime; /* time that the file was last written */
nlink_t st_nlink; /* 2 + number of subdirs, or 1 for files */
off_t st_size; /* size of file in bytes */
blksize_t st_blksize; /* block size in bytes*/
uid_t st_uid; /* what getuid() returns */
dev_t st_rdev; /* The drive number (0 = a:) */
};
```

The `st_atime`, `st_ctime` and `st_mtime` have different values only when long file names are supported (e.g. on Windows 9X); otherwise, they all have the same value: the time that the file was last written *Even when long file names are supported, the three time values returned by* `stat` *might be identical if the file was last written by a program which used legacy DOS functions that don't know about long file names.*. Most Windows 9X VFAT filesystems only support the date of the file's last access (the time is set to zero); therefore, the DJGPP implementation of `stat` sets the `st_atime` member to the same value as `st_mtime` if the time part of `st_atime` returned by the filesystem is zero (to prevent the situation where the file appears to have been created *after* it was last accessed, which doesn't look good).

The `st_size` member is an signed 32-bit integer type, so it will overflow on FAT32 volumes for files that are larger than 2GB. Therefore, if your program needs to support large files, you should treat the value of `st_size` as an unsigned value.

For some drives `st_blksize` has a default value, to improve performance. The floppy drives A: and B: default to a block size of 512 bytes. Network drives default to a block size of 4096 bytes.

Some members of `struct stat` are very expensive to compute. If your application is a heavy user of `stat` and is too slow, you can disable computation of the members your application doesn't need, as described in See _djstat_flags.

Return Value

Zero on success, nonzero on failure (and `errno` set).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
struct stat s;
stat("data.txt", &s);
if (S_ISDIR(s.st_mode))
printf("is directory\n");
```

Implementation Notes

Supplying a 100% Unix-compatible `stat` function under DOS is an implementation nightmare. The following notes describe some of the obscure points specific to `stat`s behavior in DJGPP.

1. The `drive` for character devices (like `con`, `/dev/null` and others is returned as -1. For drives networked by Novell Netware, it is returned as -2.

2. The starting cluster number of a file serves as its inode number. For files whose starting cluster number is inaccessible (empty files, files on Windows 9X, on networked drives, etc.) the `st_inode` field will be *invented* in a way which guarantees that no two different files will get the same inode number (thus it is unique). This invented inode will also be different from any real cluster number of any local file. However, only on plain DOS, and only for local, non-empty files/directories the inode is guaranteed to be consistent between `stat`, `fstat` and `lstat` function calls. (Note that two files whose names are identical but for the drive letter, will get the same invented inode, since each filesystem has its own independent inode numbering, and comparing files for identity should include the value of `st_dev`.)

3. The WRITE access mode bit is set only for the user (unless the file is read-only, hidden or system). EXECUTE bit is set for directories, files which can be executed from the DOS prompt (batch files, .com, .dll and .exe executables) or run by `go32-v2`.

4. Size of directories is reported as the number of its files (sans '.' and '..' entries) multiplied by 32 bytes (the size of directory entry). On FAT filesystems that support the LFN API (such as Windows 9X), the reported size of the directory accounts for additional space used to store the long file names.

5. Time stamp for root directories is taken from the volume label entry, if that's available; otherwise, it is reported as 1-Jan-1980.

6. The variable `_djstat_flags` (See _djstat_flags) controls what hard-to-get fields of `struct stat` are needed by the application.

7. `stat` should not be used to get an up-to-date info about a file which is open and has been written to, because `stat` will only return correct data after the file is closed. Use `fstat` (See fstat) while the file is open. Alternatively, you can call `fflush` and `fsync` to make the OS flush all the file's data to the disk, before calling `stat`.

8. The number of links `st_nlink` is always 1 for files other than directories. For directories, it is the number of subdirectories plus 2. This is so that programs written for Unix that depend on this to optimize recursive traversal of the directory tree, will still work.

statfs

Syntax

```
#include <sys/vfs.h>

int statfs(const char *filename, struct statfs *buf);
```

Description

This function returns information about the given "filesystem". The drive letter of the given filename, or the default drive if none is given, is used to retrieve the following structure:

```
struct statfs
{

long f_type; /* 0 */
long f_bsize; /* bytes per cluster */
long f_blocks; /* clusters on drive */
long f_bfree; /* available clusters */
long f_bavail; /* available clusters */
long f_files; /* clusters on drive */
long f_ffree; /* available clusters */
fsid_t f_fsid; /* array: [0]=drive_number, [1]=MOUNT_UFS */
long f_magic; /* FS_MAGIC */
};
```

Note that if INT 21h is hooked by a TSR, the total size is limited to approximately 2GB (See statvfs).

Note that there is a POSIX-compliant function `statvfs` (See statvfs), which returns similar information.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
struct statfs fs;
unsigned long long bfree, bsize;

statfs("anything", &fs);
bfree = fs.f_bfree;
bsize = fs.f_bsize;
printf("%llu bytes left\n", bfree * bsize);
```

_status87

Syntax

```
#include <float.h>

unsigned int _status87(void);
```

Description

Returns the status word of the FPU, which indicate the results of the most recently completed FPU operation:

```
---- ---- ---- ---X = SW_INVALID - invalid operation
---- ---- ---- --X- = SW_DENORMAL - denormalized operand
---- ---- ---- -X-- = SW_ZERODIVIDE - division by zero
---- ---- ---- X--- = SW_OVERFLOW - overflow
---- ---- ---X ---- = SW_UNDERFLOW - underflow
---- ---- --X- ---- = SW_INEXACT - loss of precision
---- ---- -X-- ---- = SW_STACKFAULT - stack over/under flow
---- ---- X--- ---- = SW_ERRORSUMMARY - set if any errors
-X-- -XXX ---- ---- = SW_COND - condition code
---- ---X ---- ---- = SW_C0
---- --X- ---- ---- = SW_C1
---- -X-- ---- ---- = SW_C2
-X-- ---- ---- ---- = SW_C3
--XX X--- ---- ---- = SW_TOP - top of stack (use SW_TOP_SHIFT
to shift it)
X--- ---- ---- ---- = SW_BUSY - fpu is busy
```

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

statvfs

Syntax

```
#include <sys/types.h>
#include <sys/statvfs.h>

int statvfs (const char *path, struct statvfs *sbuf);
```

Description

This function returns information about the ‘‘filesystem’’ (FS) containing path and stores it in sbuf, which has the structure below:

```
struct statvfs {
unsigned long f_bsize; /* FS block size */
unsigned long f_frsize; /* fundamental block size */
fsblkcnt_t f_blocks; /* # of blocks on FS of size f_frsize */
fsblkcnt_t f_bfree; /* # of free blocks on FS of size f_frsize */
fsblkcnt_t f_bavail; /* # of free blocks on FS of size f_frsize
* for unprivileged users */
fsfilcnt_t f_files; /* # of file serial numbers */
fsfilcnt_t f_ffree; /* # of free file serial numbers */
fsfilcnt_t f_favail; /* # of free file serial numbers
* for unprivileged users */
unsigned long f_fsid; /* FS identifier */
unsigned long f_flag; /* FS flags: bitwise OR of ST_NOSUID,
* ST_RDONLY */
unsigned long f_namemax; /* Maximum file name length on FS */
};
```

Note that if INT 21h is hooked by a TSR, the total size is limited to approximately 2GB. TSRs that hook INT 21h include:

- CD-ROM drivers;
- command-line enhancers such as CMDEDIT and Microsoft’s DOSEdit.

These may be loaded by `autoexec.bat` or `config.sys`.

The fundamental block size is considered to be a cluster. Really the fundamental block is the sector of the physical media rather than the logical block of the filesystem, but the sector size cannot be determined in all cases. So for consistency we return the cluster size.

Return Value

Zero on success, nonzero on failure (and `errno` set).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. See the comments on the fundamental block size above.

_stklen

Syntax

```
extern int _stklen;
```

Description

This variable sets the minimum stack length that the program requires. Note that the stack may be much larger than this. This value should be set statically, as it is only used at startup.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int _stklen = 256000;
```

stpcpy

Syntax

```
#include <string.h>

char *stpcpy(char *_dest, const char *_src);
```

Description

Like `strcpy` (See strcpy), but return value different.

Return Value

Returns a pointer to the trailing NUL in dest.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

stpncpy

Syntax

```
#include <string.h>

char *stpncpy(char *_dest, const char *_src, size_t _n);
```

Description

Copies exactly _n characters from _src to _dest. If need be, _dest is padded with zeros to make _n characters. Like `strncpy` (See strncpy), but return value different.

Return Value

Returns a pointer to the character following the last nonzero written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

strcasecmp

Syntax

```
#include <string.h>

int strcasecmp(const char *s1, const char *s2);
```

Description

This function compares the two strings, disregarding case.

Return Value

Zero if they're the same, nonzero if different, the sign indicates "order".

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (strcasecmp(arg, "-i") == 0)
do_include();
```

strcat

Syntax

```
#include <string.h>

char *strcat(char *s1, const char *s2);
```

Description

This function concatenates s2 to the end of s1.

Return Value

s1

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char buf[100] = "hello";
strcat(buf, " there");
```

strchr

Syntax

```
#include <string.h>

char *strchr(const char *s, int c);
```

Description

This function returns a pointer to the first occurrence of c in s. Note that if c is `NULL`, this will return a pointer to the end of the string.

Return Value

A pointer to the character, or `NULL` if it wasn't found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char *slash = strchr(filename, '/');
```

strcmp

Syntax

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

Description

This function compares s1 and s2.

Return Value

Zero if the strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequense, else a negative number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (strcmp(arg, "-i") == 0)
do_include();
```

strcoll

Syntax

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Description

This function compares s1 and s2, using the collating sequences from the current locale.

Return Value

Zero if the strings are equal, a positive number if s1 comes after s2 in the collating sequense, else a negative number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
while (strcoll(var, list[i]) < 0)
i++;
```

strcpy

Syntax

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

```
char *strcpy(char *s1, const char *s2);
```

Description

This function copies s2 into s1.

Return Value

s1

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char buf[100];
strcpy(buf, arg);
```

strcspn

Syntax

```
#include <string.h>

size_t strcspn(const char *s1, const char *set);
```

Description

This function finds the first character in s1 that matches any character in set. Note that the `NULL` bytes at the end of each string counts, so you'll at least get a pointer to the end of the string if nothing else.

Return Value

The index of the found character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int i = strcspn(command, "<>|");
if (command[i])
do_redirection();
```

strdup

Syntax

```
#include <string.h>

char * strdup (const char *source);
```

Description

Returns a newly allocated area of memory that contains a duplicate of the string pointed to by source. The memory returned by this call must be freed by the caller.

Return Value

Returns the newly allocated string, or NULL if there is no more memory.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *foo()
{

return strdup("hello");
}
```

strerror

Syntax

```
#include <string.h>

char *strerror(int error);
```

Description

This function returns a string that describes the error.

Return Value

A pointer to a static string that should not be modified or free'd.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (f=fopen("foo", "r") == 0)
printf("Error! %s: %s\n", "foo", strerror(errno));
```

strftime

Syntax

```
#include <time.h>

size_t strftime(char *buf, size_t n, const char *format,
const struct tm *time_info);
```

Description

This function formats the time data in time_info according to the given format and stores it in buf, not exceeding n bytes.

The format string is like `printf` in that any character other than `%` is added to the output string, and for each character following a `%` a pattern is added to the string as follows, with the examples as if the time was Friday, October 1, 1993, at 03:30:34 PM EDT:

`%A`
: The full weekday name (`Friday`)

`%a`
: The abbreviated weekday name (`Fri`)

`%B`
: The full month name (`October`)

`%b`
`%h`
: The abbreviated month name (`Oct`)

`%C`
: Short for `%a %b %e %H:%M:%S %Y` (`Fri Oct 1 15:30:34 1993`)

`%c`
: Short for `%m/%d/%y %H:%M:%S` (`10/01/93 15:30:34`)

`%e`
: The day of the month, blank padded to two characters (2)

`%D`
: Short for `%m/%d/%y` (10/01/93)

`%d`
: The day of the month, zero padded to two characters (02)

`%H`
: The hour (0-24), zero padded to two characters (15)

`%I`
: The hour (1-12), zero padded to two characters (03)

`%j`
: The Julian day, zero padded to three characters (275)

`%k`
: The hour (0-24), space padded to two characters (15)

`%l`
: The hour (1-12), space padded to two characters(3)

`%M`
: The minutes, zero padded to two characters (30)

`%m`
: The month (1-12), zero padded to two characters (10)

`%n`
: A newline (\n)

`%p`
: AM or PM (PM)

`%R`
: Short for `%H:%M` (15:30)

`%r`
: Short for `%I:%M:%S %p` (03:30:35 PM)

`%S`
: The seconds, zero padded to two characters (35)

`%T`
: Short for `%H:%M:%S` (15:30:35)

`%t`
: A tab (\t)

`%U`
: The week of the year, with the first week defined by the first Sunday of the year, zero padded to two characters (39)

`%u`
: The day of the week (1-7) (6)

`%W`
: The week of the year, with the first week defined by the first Monday of the year, zero padded to two characters (39)

`%w`
: The day of the week (0-6) (5)

`%x`
: Date represented according to the current locale.

%X
: Time represented according to the current locale.

%y
: The year (00-99) of the century (93)

%Y
: The year, zero padded to four digits (1993)

%Z
: The timezone abbreviation (EDT)

%%
: A percent symbol (%)

Return Value

The number of characters stored.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
time_t now = time (NULL);
struct tm *t = localtime (&now);
char buf[100];
/* Print today's date e.g. "January 31, 2001". */
strftime (buf, 100, "%B %d, %Y", t);
```

stricmp

Syntax

```
#include <string.h>

int stricmp(const char *s1, const char *s2);
```

Description

This function compares the two strings, disregarding case.

Return Value

Zero if they're the same, nonzero if different, the sign indicates "order".

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (stricmp(arg, "-i") == 0)
do_include();
```

strlcat

Syntax

```
#include <string.h>

size_t strlcat (char *dest, const char *src, size_t size);
```

Description

Concatenate characters from src to dest and nul-terminate the resulting string. As much of src is copied into dest as there is space for.

size should be the size of the destination string buffer dest plus the space for the nul-terminator. size may be computed in many cases using the `sizeof` operator.

`strlcat` may be used as a less ambiguous alternative to `strncat` (See strncat). `strlcat` returns the length of the concatenated string whether or not it was possible to copy it all --- this makes it easier to calculate the required buffer size.

If dest is not nul-terminated, then dest is not modified.

`strlcat` will not examine more than size characters of dest. This is to avoid overrunning the buffer dest.

If dest and src are overlapping buffers, the behavior is undefined. One possible result is a buffer overrun - accessing out-of-bounds memory.

The original OpenBSD paper describing `strlcat` and `strlcpy` (See strlcpy) is available on the web: http://www.openbsd.org/papers/strlcpy-paper.ps.

Return Value

The length of the string that `strlcat` tried to create is returned, whether or not `strlcat` could store it in dest. If all of src was concatenated to dst, the return value will be less than size.

If dest is not nul-terminated, then `strlcat` will consider dest to be size in length and return size plus the length of src.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

The following example shows how you can check that the destination string buffer was large enough to store the source string concatenated to the destination string. In this case `somestring` is truncated, when it is concatenated to `buf`.

```
const char somestring[] = "bar";
char buf[5] = "foo";

if (strlcat(buf, somestring, sizeof(buf)) >= sizeof(buf))
puts("somestring was truncated, when concatenating to buf.");
```

strlcpy

Syntax

```
#include <string.h>

size_t strlcpy (char *dest, const char *src, size_t size);
```

Description

Copy characters from src to dest and nul-terminate the resulting string. Up to `size - 1` characters are copied to dest.

size should be the size of the destination string buffer dest plus the space for the nul-terminator. size may be computed in many cases using the `sizeof` operator.

`strlcpy` is a less ambiguous version of `strncpy` (See strncpy). Unlike `strncpy`, `strlcpy` *always* nul-terminates the destination dest for non-zero sizes size. `strlcpy` returns the length of the string whether or not it was possible to copy it all --- this makes it easier to calculate the required buffer size.

If dest and src are overlapping buffers, the behavior is undefined. One possible result is a buffer overrun - accessing out-of-bounds memory.

The original OpenBSD paper describing `strlcpy` and `strlcat` (See strlcat) is available on the web: http://www.openbsd.org/papers/strlcpy-paper.ps.

Return Value

The length of the string that `strlcpy` tried to create is returned, whether or not `strlcpy` could store it in dest. If all of src was copied, the return value will be less than size.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

The following example shows how you can check that the destination string buffer was large enough to store the source string. In this case `somestring` is truncated to fit into `buf`.

```
const char somestring[] = "foo";
char buf[3];

if (strlcpy(buf, somestring, sizeof(buf)) >= sizeof(buf))
puts("somestring was truncated, when copying to buf.");
```

strlen

Syntax

```
#include <string.h>

size_t strlen(const char *string);
```

Description

This function returns the number of characters in string.

Return Value

The length of the string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (strlen(fname) > PATH_MAX)
invalid_file(fname);
```

strlwr

Syntax

```
#include <string.h>

char *strlwr(char *string);
```

Description

This function replaces all upper case letters in string with lower case letters.

Return Value

The string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[100] = "Hello";
strlwr(buf);
```

strncasecmp

Syntax

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

```
int strncasecmp(const char *s1, const char *s2, size_t max);
```

Description

This function compares s1 and s2, ignoring case, up to a maximum of max characters.

Return Value

Zero if the strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequense, else a negative number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (strncasecmp(foo, "-i", 2) == 0)
do_include();
```

strncat

Syntax

```
#include <string.h>

char *strncat(char *s1, const char *s2, size_t max);
```

Description

This function concatenates up to max characters of s2 to the end of s1.

Return Value

s1

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
strncat(fname, extension, 4);
```

strncmp

Syntax

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t max);
```

Description

This function compares up to max characters of s1 and s2.

Return Value

Zero if the strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequense, else a negative number.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (strncmp(arg, "-i", 2) == 0)
do_include();
```

strncpy
Syntax
```
#include <string.h>

char *strncpy(char *s1, const char *s2, size_t max);
```
Description
This function copies up to max characters of s2 into s1.

Return Value
s1

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example
```
char buf[100];
strncpy(buf, arg, 99);
```
strnicmp
Syntax
```
#include <string.h>

int strnicmp(const char *s1, const char *s2, size_t max);
```
Description
This function compares s1 and s2, ignoring case, up to a maximum of max characters.

Return Value
Zero if the strings are equal, a positive number if s1 comes after s2 in the ASCII collating sequense, else a negative number.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example
```
if (strnicmp(foo, "-i", 2) == 0)
do_include();
```
strpbrk
Syntax
```
#include <string.h>

char *strpbrk(const char *s1, const char *set);
```
Description
This function finds the first character in s1 that matches any character in set.

Return Value
A pointer to the first match, or `NULL` if none are found.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (strpbrk(command, "<>|"))
do_redirection();
```

strrchr

Syntax

```
#include <string.h>

char *strrchr(const char *s1, int c);
```

Description

This function finds the last occurrence of `c` in `s1`.

Return Value

A pointer to the last match, or `NULL` if the character isn't in the string.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char *last_slash = strrchr(filename, '/');
```

strsep

Syntax

```
#include <string.h>

char *strsep(char **stringp, char *delim);
```

Description

This function retrieves the next token from the given string, where stringp points to a variable holding, initially, the start of the string. Tokens are delimited by a character from delim. Each time the function is called, it returns a pointer to the next token, and sets *stringp to the next spot to check, or `NULL`.

Return Value

The next token, or NULL.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
main()
{

char *buf = "Hello there,stranger";
char **bp = &buf;
char *tok;
while (tok = strsep(bp, " ,"))
printf("tok = ‘%s’\n", tok);
}


tok = ‘Hello’
tok = ‘’
tok = ‘there’
tok = ‘stranger’
```

strspn

Syntax

```
#include <string.h>

size_t strspn(const char *s1, const char *set);
```

Description

This function finds the first character in s1 that does not match any character in set. Note that the `NULL` bytes at the end of s1 counts, so you'll at least get a pointer to the end of the string if nothing else.

Return Value

The index of the found character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int i = strspn(entry, " \t\b");
if (entry[i])
do_something();
```

strstr

Syntax

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

Description

This function finds the first occurrence of s2 in s1.

Return Value

A pointer within s1, or `NULL` if s2 wasn't found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
if (strstr(command, ".exe"))
do_exe();
```

strtod

Syntax

```
#include <stdlib.h>

double strtod(const char *s, char **endp);
```

Description

This function converts as many characters of s as look like a floating point number into that number. It also recognises (case-insensitively) ''Inf'', ''Infinity'', ''NaN'', ''NaN(optional decimal-number)'', ''NaN(optional octal-number)'' and ''NaN(optional hex-number)''. If endp is not a null pointer, a pointer to the first unconverted character will be stored in the location pointed to by endp.

Return Value

The value represented by s.

If s is ‘‘Inf’’ or ‘‘Infinity’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `INFINITY` (if no prefix or a ‘‘+’’ prefix) or `-INFINITY` (if the prefix is ‘‘-’’).

If s is ‘‘NaN’’ or ‘‘NaN()’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `(double)NAN`. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1.

If s is ‘‘NaN(decimal-number)’’, ‘‘NaN(octal-number)’’ or ‘‘NaN(hex-number)’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is a NaN with the mantissa bits set to the lower 52 bits of decimal-number, octal-number or hex-number (the mantissa for doubles consists of 52 bits). Use at most 16 hexadecimal digits in hex-number or the internal conversion will overflow, which results in a mantissa with all bits set. If the bit pattern given is 0 (which won’t work as a representation of a NaN) `(double)NAN` will be returned. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1. Testing shows that SNaNs might be converted into QNaNs (most significant bit will be set in the mantissa).

If a number represented by s doesn’t fit into the range of values representable by the type `double`, the function returns either `-HUGE_VAL` (if s begins with the character `-`) or `+HUGE_VAL`, and sets `errno` to `ERANGE`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1) 1003.2-1992; 1003.1-2001

Notes:

1. Support for ‘‘Inf’’, ‘‘Infinity’’, ‘‘NaN’’ and ‘‘NaN(…)’’ was standardised in ANSI C99.

Example

```
char buf[] = "123ret";
char buf2[] = "0x123ret";
char buf3[] = "NAN(123)";
char buf4[] = "NAN(0x123)";
char *bp;
double x, x2, x3, x4;

x = strtod(buf, &bp);
x2 = strtod(buf2, &bp);
x3 = strtod(buf3, &bp);
x4 = strtod(buf4, &bp);
```

strtof

Syntax

```
#include <stdlib.h>

float strtof(const char *s, char **endp);
```

Description

This function converts as many characters of s as look like a floating point number into that number. It also recognises (case-insensitively) ‘‘Inf’’, ‘‘Infinity’’, ‘‘NaN’’, ‘‘NaN(optional decimal-number)’’, ‘‘NaN(optional octal-number)’’ and ‘‘NaN(optional hex-number)’’. If endp is not a null pointer, a pointer to the first unconverted character will be stored in the location pointed to by endp.

Return Value

The value represented by s.

If s is ‘‘Inf’’ or ‘‘Infinity’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `INFINITY` (if no prefix or a ‘‘+’’ prefix) or `-INFINITY` (if the prefix is ‘‘-’’).

If s is ‘‘NaN’’ or ‘‘NaN()’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `NAN`. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1.

If s is ‘‘NaN(decimal-number)’’, ‘‘NaN(octal-number)’’ or ‘‘NaN(hex-number)’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is a NaN with the mantissa bits set to the lower 23 bits of decimal-number, octal-number or hex-number (the mantissa for floats consists of 23 bits). Use at most 8 hexadecimal digits in hex-number or the internal conversion will overflow, which results in a mantissa with all bits set. If the bit pattern given is 0 (which won't work as a representation of a NaN) `NAN` will be returned. If the prefix is ''-'' the sign bit in the NaN will be set to 1. Testing shows that SNaNs might be converted into QNaNs (most significant bit will be set in the mantissa).

If a number represented by s doesn't fit into the range of values representable by the type `float`, the function returns either `-HUGE_VALF` (if s begins with the character `-`) or `+HUGE_VALF`, and sets `errno` to `ERANGE`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99 (see note 1); not C89 1003.1-2001; not 1003.2-1992

Notes:

1. Support for ''Inf'', ''Infinity'', ''NaN'' and ''NaN(…)'' was standardised in ANSI C99.

Example

```
char buf[] = "123ret";
char buf2[] = "0x123ret";
char buf3[] = "NAN(123)";
char buf4[] = "NAN(0x123)";
char *bp;
float x, x2, x3, x4;

x = strtof(buf, &bp);
x2 = strtof(buf2, &bp);
x3 = strtof(buf3, &bp);
x4 = strtof(buf4, &bp);
```

strtoimax

Syntax

```
#include <inttypes.h>

intmax_t strtoimax (const char *s, char **endp, int base)
```

Description

This function converts as much of s as looks like an appropriate number into the value of that number, and sets *endp to point to the first unused character.

The base argument indicates what base the digits (or letters) should be treated as. If base is zero, the base is determined by looking for `0x`, `0X`, or `0` as the first part of the string, and sets the base used to 16, 16, or 8 if it finds one. The default base is 10 if none of those prefixes are found.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %" PRIdMAX "\n", strtoimax(buf, &bp, 0));
```

strtok

Syntax

```
#include <string.h>

char *strtok(char *s1, const char *s2);
```

Description

This function retrieves tokens from s1 which are delimited by characters from s2.

To initiate the search, pass the string to be searched as s1. For the remaining tokens, pass `NULL` instead.

Return Value

A pointer to the token, or `NULL` if no more are found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
main()
{

char *buf = "Hello there, stranger";
char *tok;
for (tok = strtok(buf, " ,");
tok;
tok=strtok(0, " ,"))
printf("tok = ‘%s’\n", tok);
}


tok = ‘Hello’
tok = ‘there’
tok = ‘stranger’
```

strtol

Syntax

```
#include <stdlib.h>

long strtol(const char *s, char **endp, int base);
```

Description

This function converts as much of s as looks like an appropriate number into the value of that number. If endp is not a null pointer, *endp is set to point to the first unused character.

The base argument indicates what base the digits (or letters) should be treated as. If base is zero, the base is determined by looking for `0x`, `0X`, or `0` as the first part of the string, and sets the base used to 16, 16, or 8 if it finds one. The default base is 10 if none of those prefixes are found.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %d\n", strtol(buf, &bp, 0));
```

strtold

Syntax

```
#include <stdlib.h>

long double strtold(const char *s, char **endp);
```

Description

This function converts as many characters of s that look like a floating point number into that number. It also recognises (case-insensitively) ‘‘Inf’’, ‘‘Infinity’’, ‘‘NaN’’, ‘‘NaN(optional decimal-number), ‘‘NaN(optional octal-number) and ‘‘NaN(optional hex-number)’’. If endp is not a null pointer, a pointer to the first unconverted character will be stored in the location pointed to by endp.

Return Value

The value represented by s.

If s is ‘‘Inf’’ or ‘‘Infinity’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `INFINITY` (if no prefix or a ‘‘+’’ prefix) or `-INFINITY` (if the prefix is ‘‘-’’).

If s is ‘‘NaN’’ or ‘‘NaN()’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `(long double)NAN`. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1.

If s is ‘‘NaN(decimal-number)’’, ‘‘NaN(octal-number)’’ or ‘‘NaN(hex-number)’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is a NaN with the mantissa bits set to the lower 63 bits of decimal-number, octal-number or hex-number and the most significant bit to 1 (the mantissa for long doubles consists of 64 bits where the most significant bit is the integer bit which must be set for NaNs). Use at most 16 hexadecimal digits in hex-number or the internal conversion will overflow, which results in a mantissa with all bits set. If the bit pattern given is 0 (which won’t work as a representation of a NaN) `(long double)NAN` will be returned. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1. Testing shows that SNaNs might be converted into QNaNs (the second most significant bit will be set in the mantissa).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99 (see note 1); not C89
1003.1-2001; not 1003.2-1992

Notes:

1. Support for ‘‘Inf’’, ‘‘Infinity’’, ‘‘NaN’’ and ‘‘NaN(…)’’ was standardised in ANSI C99.

Example

```
char buf[] = "123ret";
char buf2[] = "0x123ret";
char buf3[] = "NAN(123)";
char buf4[] = "NAN(0x123)";
char *bp;
long double x, x2, x3, x4;

x = strtold(buf, &bp);
x2 = strtold(buf2, &bp);
x3 = strtold(buf3, &bp);
x4 = strtold(buf4, &bp);
```

_strtold

Syntax

```
#include <stdlib.h>

long double _strtold(const char *s, char **endp);
```

Description

This function converts as many characters of s that look like a floating point number into that number. It also recognises (case-insensitively) ‘‘Inf’’, ‘‘Infinity’’, ‘‘NaN’’, ‘‘NaN(optional decimal-number)’’, ‘‘NaN(optional octal-number)’’ and ‘‘NaN(optional hex-number)’’. If endp is not a null pointer, a pointer to the first unconverted character will be stored in the location pointed to by endp.

There is also a standardised version of this function: `strtold` (See strtold).

Return Value

The value representedby s.

If s is ‘‘Inf’’ or ‘‘Infinity’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `INFINITY` (if no prefix or a ‘‘+’’ prefix) or `-INFINITY` (if the prefix is ‘‘-’’).

If s is ‘‘NaN’’ or ‘‘NaN()’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is `(long double)NAN`. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1.

If s is ‘‘NaN(decimal-number)’’, ‘‘NaN(octal-number)’’ or ‘‘NaN(hex-number)’’, with any variations of case and optionally prefixed with ‘‘+’’ or ‘‘-’’, the return value is a NaN with the mantissa bits set to the lower 63 bits of decimal-number, octal-number or hex-number and the most significant bit to 1 (the mantissa for long doubles consists of 64 bits where the most significant bit is the integer bit which must be set for NaNs). Use at most 16 hexadecimal digits in hex-number or the internal conversion will overflow, which results in a mantissa with all bits set. If the bit pattern given is 0 (which won’t work as a representation of a NaN) `(long double)NAN` will be returned. If the prefix is ‘‘-’’ the sign bit in the NaN will be set to 1. Testing shows that SNaNs might be converted into QNaNs (the second most significant bit will be set in the mantissa).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[] = "123ret";
char buf2[] = "0x123ret";
char buf3[] = "NAN(123)";
char buf4[] = "NAN(0x123)";
char *bp;
long double x, x2, x3, x4;

x = _strtold(buf, &bp);
x2 = _strtold(buf2, &bp);
x3 = _strtold(buf3, &bp);
x4 = _strtold(buf4, &bp);
```

strtoll

Syntax

```
#include <stdlib.h>

long long int strtoll(const char *s, char **endp, int base);
```

Description

This function converts as much of s as looks like an appropriate number into the value of that number, and sets *endp to point to the first unused character.

The base argument indicates what base the digits (or letters) should be treated as. If base is zero, the base is determined by looking for `0x`, `0X`, or `0` as the first part of the string, and sets the base used to 16, 16, or 8 if it finds one. The default base is 10 if none of those prefixes are found.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %lld\n", strtoll(buf, &bp, 0));
```

strtoul

Syntax

```
#include <stdlib.h>

unsigned long strtoul(const char *s, char **endp, int base);
```

Description

This is just like `strtol` (See strtol) except that the result is unsigned.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %u\n", strtoul(buf, &bp, 0));
```

strtoull

Syntax

```
#include <stdlib.h>

unsigned long long int strtoull(const char *s, char **endp, int base);
```

Description

This is just like `strtoll` (See strtoll) except that the result is unsigned.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %llu\n", strtoull(buf, &bp, 0));
```

strtoumax

Syntax

```
#include <inttypes.h>

uintmax_t strtoumax (const char *s, char **endp, int base);
```

Description

This is just like `strtoimax` (See strtoimax) except that the result is unsigned.

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C99; not C89 1003.1-2001; not 1003.2-1992

Example

```
printf("Enter a number: "); fflush(stdout);
gets(buf);
char *bp;
printf("The value is %" PRIuMAX "\n", strtoumax(buf, &bp, 0));
```

strupr

Syntax

```
#include <string.h>

char *strupr(char *string);
```

Description

This function converts all lower case characters in string to upper case.

Return Value

string

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char buf[] = "Foo!";
strupr(buf);
```

strxfrm

Syntax

```
#include <string.h>

size_t strxfrm(char *s1, const char *s2, size_t max);
```

Description

This copies characters from s2 to s1, which must be able to hold max characters. Each character is transformed according to the locale such that `strcmp(s1b, s2b)` is just like `strcoll(s1, s2)` where `s1b` and `s2b` are the transforms of `s1` and `s2`.

Return Value

The actual number of bytes required to transform s2, including the `NULL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

swab

Syntax

```
#include <stdlib.h>

void swab(const void *from, void *to, int nbytes);
```

Description

This function copies nbytes bytes from the address pointed to by from to the address pointed by to, exchanging adjacent even and odd bytes. It is useful for carrying binary data between little-endian and big-endian machines.

The argument nbytes should be even, and the buffers from and to should not overlap.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

symlink

Syntax

```
#include <unistd.h>

int symlink(const char *exists, const char *new);
```

Description

DOS does not support symbolic links. However, DJGPP emulates them---this function creates a file with special size and format, so other DJGPP library functions transparently work with file which is pointed to by the symlink. Of course, it does not work outside DJGPP programs. Those library functions which are simple wrappers about DOS calls do not use symlinks neither.

Return Value

Zero in case of success, -1 in case of failure (and `errno` set to the appropriate error code).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
symlink ("c:/djgpp/bin/grep", "c:/djgpp/bin/fgrep");
```

syms_init

Syntax

```
#include <debug/syms.h>

void syms_init (char *file);
```

Description

This function reads debugging symbols from the named file, which should be an executable program (either a `.exe` file or a raw COFF image created by `ld.exe`, the linker). It then processes the symbols: classifies them by type, sorts them by name and value, and stores them in internal data structures used by other symbol-related functions, such as `syms_val2name`, `syms_val2line`, etc.

You **must** call `syms_init` before calling the other `syms_*` functions.

Currently, `syms_init` only supports COFF and AOUT debugging format, so programs compiled with `-gstabs` cannot be processed by it.

Return Value

None.

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
syms_init("c:/foo/bar/baz.exe");
```

syms_line2val

Syntax

```
#include <debug/syms.h>

unsigned long syms_line2val (char *filename, int lnum);
```

Description

This function returns the address of the first instruction produced from the line lnum of the source file filename that was linked into a program whose symbols were read by a previous call to `syms_init`.

COFF debugging format does not support pathnames, so filename should not include leading directories, just the basename.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

Return Value

The address of the first instruction produced from the line, or zero if filename is not found in the symbol table or if no executable code was generated for line lnum in filename.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
syms_init ("foo.exe");
printf ("Line 3 of foo.c is at address %lx\n",
syms_line2val("foo.c", 3));
```

syms_listwild

Syntax

```
#include <debug/syms.h>

void syms_listwild (char *pattern,
void (*handler) (unsigned long addr, char type_c,
char *name, char *file, int lnum));
```

Description

This function walks through all the symbols that were read by a previous call to `syms_init` (See syms_init). For each symbol whose name matches pattern, it invokes the user-defined function handler, passing it information about that symbol:

`address`
the address of the symbol.

`type_c`
a letter that specifies the type of the symbol, as follows:

`T`
`t`
‘‘text’’, or code: usually a function.

`D`
`d`
data: an initialized variable.

`B`
`b`
‘‘bss’’: an uninitialized variable.

`F`
`f`
a function (in `a.out` file only).

`V`

`v`
: a set element or pointer (in `a.out` file only).

`I`
`i`
: an indirect symbol (in `a.out` file only).

`U`
`u`
: an undefined (a.k.a. unresolved) symbol.

`A`
`a`
: an absolute symbol.

`name`
: the name of the symbol.

`file`
: the source file name where the symbol is defined.

`lnum`
: the line number on which the symbol is defined in the source file.

Since variables and functions defined in C get prepended with an underscore _, begin pattern with _ if you want it to match C symbols.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
void print_sym (unsigned long addr, char type_c,
char *name, char *file, int lnum)
{
printf (file ? "%s: %lx %c %s:%d\n" : "%s: %lx %c\n",
name, addr, type_c,
file ? file : "", lnum );
}

int main (void)
{
syms_init ("foo.exe");
/* List all the symbols which begin with "___djgpp". */
syms_listwild ("___djgpp*", print_sym);
return 0;
}
```

syms_module

Syntax

```
#include <debug/syms.h>

char *syms_module (int nfile);
```

Description

This function returns the name of the source file (a.k.a. module) whose ordinal number in the symbol table is nfile.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

Return Value

The name of the source file, or a `NULL` pointer if nfile is negative or larger than the total number of modules linked into the program whose symbols were read by `syms_init`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

syms_name2val

Syntax

```
#include <debug/syms.h>
extern int undefined_symbol;
extern int syms_printwhy;

unsigned long syms_name2val (const char *string);
```

Description

This function returns the address of a symbol specified by string. string may be one of the following:

- A number, with or without a sign, in which case the number specifies the address or an offset from an address.
- A file name and a line number: `file#[line]`, where file is the name of one of the source files linked into the program whose symbols were read by `syms_init`, and line is a line number in that file. If line is omitted, it defaults to zero.

  Note that the COFF format supported by DJGPP only stores the basename of the source files, so do not specify file with leading directories.
- A symbol name as a string. The name can be specified either with or without the leading underscore _.
- A register name `%reg`. reg specifies the value of one of the debuggee's registers saved in the external variable `a_tss` (See run_child).
- Any sensible combination of the above elements, see the example below.

`syms_name2val` looks up the specified file, line, and symbol in the symbol table prepared by `syms_init`, finds their addresses, adds the offset, if any, and returns the result.

If the specified file, line, or symbol cannot be found, `syms_name2val` returns zero and sets the global variable `undefined_symbol` to a non-zero value. If the global variable `syms_printwhy` is non-zero, an error message is printed telling which part of the argument string was invalid.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

Return Value

The address specified by string, or zero, if none found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned long addr1, addr2, addr3;

syms_init ("foo.exe");
addr1 = syms_name2val ("foo.c#256+12");
addr2 = syms_name2val ("_main");
addr3 = syms_name2val ("struct_a_var+%eax+4");
```

syms_val2line

Syntax

```
#include <debug/syms.h>

char *syms_val2line (unsigned long addr, int *line, int exact);
```

Description

This function takes an address addr and returns the source file name which correspond to that address. The line number in that source file is stored in the variable pointed by line. If exact is non-zero, the function succeeds only if addr is the first address which corresponds to some source line.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

Return Value

The name of the source file which corresponds to addr, or `NULL` if none was found.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
int lineno;
char *file_name;
syms_init ("foo.exe");
file_name = syms_val2line (0x1c12, &lineno);
printf ("The address %x is on %s, line %d\n",
0x1c12, file_name, line);
```

syms_val2name

Syntax

```
#include <debug/syms.h>

char *syms_val2name (unsigned long addr, unsigned long *offset);
```

Description

This function takes an address addr and returns the name of the closest symbol whose address is less that addr. If offset is not a `NULL` pointer, the offset of addr from the symbol's address is stored in the variable pointed to by offset.

You must call `syms_init` (See syms_init) before calling any of the other `syms_*` functions for the first time.

This function is meant to be used to convert numerical addresses into function names and offsets into their code, like what `symify` does with the call frame traceback.

The function ignores several dummy symbols, like `_end` and `_etext`.

Return Value

The name of the found symbol, or the printed hexadecimal representation of addr, if no symbol was found near addr. The return value is a pointer to a static buffer, so don't overwrite it and don't pass it to `free`!

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
unsigned long offs;
char *symbol_name;
syms_init ("foo.exe");
symbol_name = syms_val2name (0x1c12, &offs);
printf ("The address %x is at %s%+ld\n", 0x1c12, symbol_name, offs);
```

sync

Syntax

```
#include <unistd.h>

int sync(void);
```

Description

Intended to assist porting Unix programs. Under Unix, `sync` flushes all caches of previously written data. In this implementation, `sync` calls `fsync` on every open file. See fsync. It also calls `_flush_disk_cache` (See _flush_disk_cache) to try to force cached data to the disk.

Return Value

Always returns 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
sync();
```

sys_errlist

Syntax

```
#include <errno.h>

extern char *sys_errlist[];
```

Description

This array contains error messages, indexed by `errno`, that describe the errors.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("Error: %s\n", sys_errlist[errno]);
```

sys_nerr

Syntax

```
#include <errno.h>

extern int sys_nerr;
```

Description

This variable gives the number of error messages in `sys_errlist` (See sys_errlist).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
if (errno < sys_nerr)
printf("Error: %s\n", sys_errlist[errno]);
```

sysconf

Syntax

```
#include <unistd.h>

long sysconf(int which);
```

Description

This function returns various system configuration values, based on which:

```
case _SC_ARG_MAX: return _go32_info_block.size_of_transfer_buffer;
case _SC_CHILD_MAX: return CHILD_MAX;
case _SC_CLK_TCK: return CLOCKS_PER_SEC;
case _SC_NGROUPS_MAX: return NGROUPS_MAX;
case _SC_OPEN_MAX: return 255;
case _SC_JOB_CONTROL: return -1;
case _SC_SAVED_IDS: return -1;
case _SC_STREAM_MAX: return _POSIX_STREAM_MAX;
case _SC_TZNAME_MAX: return TZNAME_MAX;
case _SC_VERSION: return _POSIX_VERSION;
```

Return Value

The value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

system

Syntax

```
#include <stdlib.h>

int system(const char *cmd);
```

Description

This function runs the command or program specified by cmd. If cmd is a null pointer, `system` returns non-zero only if a shell is available. If cmd is an empty string, the command processor pointed to by `SHELL` or `COMSPEC` variables in the environment will be invoked interactively; type **exit RET** to return to the program which called `system`. (Note that some other DOS compilers treat a null pointer like an empty command line, contrary to ANSI C requirements.)

When calling programs compiled by DJGPP this function will not use `COMMAND.COM` and so will not be subject to its 126 character limit on command lines.

When the calling program runs on Windows 9X or Windows 2000 and calls the system shell to run the child program, or if the child program is a native Windows program (in `PE-COFF` format), or when the system shell is `4DOS` or `NDOS`, command lines longer than 126 characters are passed via the environment variable `CMDLINE`.

Command lines and pipes (i.e., the use of `<`, `>`, `>>`, and `|`) will be simulated internally in this function; this means that you can have both long command lines and redirection/pipes when running DJGPP programs with this function.

By default, `COMMAND.COM` will only be invoked to run commands internal to it, or to run batch files (but this can be changed, see below). In these cases, the returned error code will always be zero, since `COMMAND.COM` always exits with code 0.

Certain commands internal to `COMMAND.COM` that don't make sense or cause no effect in the context of `system` are ignored by this function. These are `REM`, `EXIT`, `GOTO`, `SHIFT`; `SET`, `PATH` and `PROMPT` are ignored only if called with an argument. You can disable this feature if you need, see below.

Some commands are emulated internally by `system`, because the emulation is better than the original. Currently, the only emulated command is `CD` or `CHDIR`: the emulation knows about forward slashes and also switches the current drive. This emulation can also be switched off, as explained below.

When `system` is presented with an internal shell command, it checks the environment variables `SHELL` and `COMSPEC` (in that order) and invokes the program that they point to. If the shell thus found is one of the DOS shells (`COMMAND.COM`, `4DOS` or `NDOS`), they are called with the `/c` switch prepended to the command line. Otherwise, `system` assumes that the shell is a Unix-style shell and passes it the entire command line via a temporary file, invoking the shell with a single argument which is the name of that file.

Shell scripts and batch files are invoked by calling either the program whose name appears on the first line (like in `#! /bin/sh`), or the default shell if none is specified by the script. If the name of the shell specified by the script is a Unix-style pathname, without a drive letter and with no extension, `system` will additionally search for it on the `PATH`. This allows to invoke Unix shell scripts unmodified, if you have a ported shell installed on your system.

You can customize the behavior of `system` using a bit-mapped variable `__system_flags`, defined on `<stdlib.h>`. The following bits are currently defined:

`__system_redirect`
: When set (the default), specifies that `system` can use its internal redirection and pipe code. If reset, any command line that includes an unquoted redirection symbol will be passed to the shell.

`__system_call_cmdproc`
: When set, `system` will always call the shell to execute the command line. If reset (the default), the shell will only be called when needed, as described above.

  You should *always* set this bit if you use a real, Unix-style shell (also, set `__system_use_shell`, described below, and the `SHELL` environment variable).

`__system_use_shell`
: When set (the default), the `SHELL` environment variable will take precedence upon `COMSPEC`; this allows you to specify a special shell for `system` that doesn't affect the rest of DOS. If reset, only `COMSPEC` is used to find the name of the command processor.

`__system_allow_multiple_cmds`
: When set, you can put multiple commands together separated by the `;` character. If reset (the default), the command line passed to `system` is executed as a single command and `;` has no special meaning.

`__system_allow_long_cmds`
: When set (the default), `system` will handle command lines longer than the DOS 126-character limit; this might crash your program in some cases, as the low-level functions that invoke the child program will only pass them the first 126 characters. When reset, `system` will detect early that the command line is longer than 126 characters and refuse to run it, but you will not be able to call DJGPP programs with long command lines.

`__system_emulate_command`
: If reset (the default), `system` will pass the entire command line to the shell if its name is one of the following: `sh.exe`, `sh16.exe`, `sh32.exe`, `bash.exe`, `tcsh.exe`. When set, `system` will attempt to emulate redirection and pipes internally, even if `COMSPEC` or `SHELL` point to a Unix-style shell.

`__system_handle_null_commands`
: When set (the default), commands internal to `COMMAND.COM` and compatible shells which have no effect in the context of `system`, are ignored (the list of these commands was given above). If reset, these commands are processed as all others, which means `COMMAND.COM` will be called to execute them.

  Note that this bit shouldn't be used with a Unix-style shell, because it does the wrong thing then. With Unix-style shells, you are supposed to set the `__system_call_cmdproc` bit which will always call the shell.

`__system_ignore_chdir`
: If set, the `CD` and `CHDIR` commands are ignored. When reset (the default), the processing of these commands depends on the `__system_emulate_chdir` bit, see below.

  This bit is for compatibility with Unix, where a single `cd dir` command has no effect, because the current working directory there is not a global notion (as on MSDOS). Don't set this bit if you use multiple commands (see `__system_allow_multiple_cmds` above).

`__system_emulate_chdir`
: When set, the `CD` and `CHDIR` commands are emulated internally: they change the drive when the argument specifies a drive letter, and they support both forward slashes and backslashes in pathnames. When `CD` is called without an argument, it prints the current working directory with forward slashes and down-cases DOS 8+3 names. If this bit is reset (the default), `CD` and `CHDIR` are passed to the shell.

The behavior of `system` can be customized at run time by defining the variable `DJSYSFLAGS` in the environment. The value of that variable should be the numerical value of `__system_flags` that you'd like to set; it will override the value of `__system_flags` specified when the program was compiled.

Return Value

If cmd is a null pointer, `system` returns non-zero if a shell is available. The actual test for the existence of an executable file pointed to by `SHELL` or `COMSPEC` is only performed if the shell is to be invoked to process the entire command line; if most of the work is to be done by `system` itself, passing a null pointer always yields a non-zero return value, since the internal emulation is always ‘‘available’’.

Otherwise, the return value is the exit status of the child process in its lower 8 bits; bits 8-17 of the return value will hold `SIGINT` or `SIGABRT` if the child process was aborted by **Ctrl-C** or Critical Device Error, respectively; otherwise they will be zero *Many DOS programs catch* **Ctrl-C** *keystrokes and Critical Errors, and handle them in customized ways. If this handling prevents DOS from realizing that the program was aborted due to these reasons, bits 8-17 of the value returned by* `system` *will most probably be zero. Don’t count on these bits to hold the signal number!*. If the child couldn’t be run, `system` will return -1 and set `errno` to an appropriate value. Note that if `COMMAND.COM` was used to run the child, it will always return a 0 status, even if the command didn’t run successfully. However, `system` only calls `COMMAND.COM` when it needs to run commands internal to it.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
system("cc1plus.exe @cc123456.gp");
```

tan

Syntax

```
#include <math.h>

double tan(double x);
```

Description

This function computes the tangent of x (which should be given in radians).

Return Value

The tangent of x. If the absolute value of x is finite but greater than or equal to 2^63, the return value is 0 (since for arguments that large each bit of the mantissa is more than `Pi`). If the value of x is infinite or `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

tanh

Syntax

```
#include <math.h>

double tanh(double x);
```

Description

This function computes the hyperbolic tangent of x.

Return Value

The hyperbolic tangent of x. If x is either a positive or a negative infinity, the result is unity with the same sign as x, and `errno` is not changed. If x is `NaN`, the return value is `NaN` and `errno` is set to `EDOM`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

tcdrain

Syntax

```
#include <termios.h>

int tcdrain (int fd);
```

Description

This function waits until all the output is written to the file/device referred to by the handle fd. In this implementation, this function does nothing except checking the validity of its arguments; it is provided for compatibility only. Note that the termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tcflow

Syntax

```
#include <termios.h>

int tcflow (int fd, int action);
```

Description

This function suspends transmission of data to, or reception of data from, the device/file open on handle fd. The action argument can take one of these values:

`TCOOFF`
: the output is suspended

`TCOON`
: the output is resumed

`TCIOFF`
: the STOP character is transmitted

`TCION`
: the START character is transmitted

The current START and STOP characters are stored in the `termios` structure that is currently in effect. See Termios functions, for more details about that.

Note that the DJGPP termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tcflush

Syntax

```
#include <termios.h>

int tcflush (int fd, int which);
```

Description

This function clears the input and/or output queues on for the device/file open on handle fd. The which argument can take these values:

`TCIFLUSH`
: the unprocessed characters in the input buffer are discarded

`TCOFLUSH`
: no effect (provided for compatibility)

`TCIOFLUSH`
: has the combined effect of `TCIFLUSH` and `TCOFLUSH`

Note that the termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tcgetattr

Syntax

```
#include <termios.h>

int tcgetattr (int fd, struct termios *termiosp);
```

Description

This function gets the parameters associated with the file/device referred to by the handle fd and stores them in the termios structure termiosp. See Termios functions, for the full description of `struct termios` and its members.

Note that the termios emulation handles console only.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
struct termios termiosbuf;
int rc = tcgetattr (0, &termiosbuf);
```

tcgetpgrp

Syntax

```
#include <termios.h>

int tcgetpgrp (int fd);
```

Description

This function returns the value of the process group ID for the foreground process associated with the terminal. The file descriptor fd must be connected to the terminal, otherwise the function will fail.

Return Value

If fd is connected to the terminal, the function returns the process group ID, which is currently identical to the value returned by `getpgrp()` (See getpgrp). Otherwise, it returns -1 and sets `errno` to `ENOTTY`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tcsendbreak

Syntax

```
#include <termios.h>

int tcsendbreak (int fd, int duration);
```

Description
This function generates a break condition for `duration*0.25` seconds. In the current implementation this function does nothing; it is provided for compatibility only. Note that the termios emulation handles console only.

Return Value
Zero on success, nonzero on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tcsetattr
Syntax
```
#include <termios.h>

int tcsetattr (int fd, int action, const struct termios *termiosp);
```

Description
This function sets termios structure for device open on the handle fd from the structure termiosp. Note that the termios emulation handles console only.

The action argument can accept the following values:

```
TCSANOW
TCSADRAIN
TCSAFLUSH
```

Currently, any of these values causes the values in termiosp to take effect immediately.

See Termios functions, for the description of the `struct termios` structure.

Return Value
Zero on success, nonzero on failure.

Portability
{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example
```
tcsetattr (0, TCSANOW, &termiosbuf);
```

tcsetpgrp
Syntax
```
#include <termios.h>

int tcsetpgrp (int fd, pid_t pgroup_id);
```

Description
This function sets the foreground process group ID for the terminal connected to file descriptor fd. fd must be a valid handle connected to a terminal device, and pgroup_id must be the process group ID of the calling process, or the function will fail.

Return Value
If fd is a valid handle connected to a terminal and pgroup_id is equal to what `getpgrp()` returns (See getpgrp), the function will do nothing and return zero. Otherwise, -1 will be returned and `errno` will be set to a suitable value. In particular, if the pgroup_id argument is different from what `getpgrp()` returns, `tcsetpgrp` sets `errno`

to ENOSYS.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tell

Syntax

```
#include <io.h>

off_t tell(int file);
```

Description

This function returns the location of the file pointer for file.

Return Value

The file pointer, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
off_t q = tell(fd);
```

telldir

Syntax

```
#include <dirent.h>

long telldir(DIR *dir);
```

Description

This function returns a value which indicates the position of the pointer in the given directory. This value is only useful as an argument to `seekdir` (See seekdir).

Return Value

The directory pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
DIR *dir;
long q = telldir(dir);
do_something();
seekdir(dir, q);
```

tempnam

Syntax

```
#include <stdio.h>

char * tempnam(const char *tmpdir, const char *prefix);
```

Description

This function generates a file name which can be used for a temporary file, and makes sure that no other file by that name exists.

The caller has control on the choice of the temporary file's directory, and the initial letters of the file's basename. If the argument tmpdir points to the name of the directory in which the temporary file will be created, `tempnam` will ensure that the generate name is unique **in that directory**. If the argument prefix points to a string, then that string will be used as the first few characters of the file's basename. Due to limitations of the DOS 8.3 file namespace, only up to two first characters in prefix will be used.

If tmpdir is `NULL`, or empty, or points to a non-existent directory, `tempnam` will use a default directory. The default directory is determined by testing, in sequence, the directories defined by the values of environment variables `TMPDIR`, `TEMP` and `TMP`. The first variable that is found to point to an existing directory will be used. If none of these variables specify a valid directory, `tempnam` will use the static default path prefix defined by `P_tmpdir` on `<stdio.h>`, or `"c:/"`, in that order.

If prefix is `NULL` or empty, `tempnam` will supply its own default prefix `"tm"`.

`tempnam` puts the generated name into space allocated by `malloc`. It is up to the caller to free that space when it is no longer needed.

Note that `tempnam` does not actually create the file, nor does it ensure in any way that the file will be automatically deleted when it's no longer used. It is the user's responsibility to do that.

Return Value

On success, `tempnam` returns a pointer to space (allocated with a call to `malloc`) where the file name is constructed. If `malloc` failed to provide sufficient memory buffer, or if no valid directory to hold the file was found, `tempnam` returns a `NULL` pointer.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <stdio.h>

tempnam ("c:/tmp/", "foo");
```

Termios functions

The `termios` functions allow to control terminals and asynchronous communications ports. The DJGPP implementation currently supports the `termios` functionality for console devices only. It does that by reading the keyboard via the BIOS Int 16h and writes to the screen via the direct output interrupt 29h. This I/O redirection is performed by the special hook internal to the library.

Many of the `termios` functions accept a termiosp argument which is a pointer to a `struct termios` variable. Here's the description of this structure:

```
#define NCCS 12
struct termios {
cc_t c_cc[NCCS]; /* control characters */
tcflag_t c_cflag; /* control modes */
tcflag_t c_iflag; /* input modes */
tcflag_t c_lflag; /* local modes */
tcflag_t c_oflag; /* output modes */
speed_t c_ispeed; /* input baudrate */
speed_t c_ospeed; /* output baudrate */
}
```

The array `c_cc[]` defines the special control characters. the following table lists the supported control functions the default characters which invoke those functions, and the default values for MIN and TIME parameters:

{Index {VERASE {Delete previous character {Backspace

The special characters (like `VEOL`, `VKILL`, etc.) produce their effect only under the canonical input processing, that is, when the `ICANON` bit in the `c_lflag` member of `struct termios` (see below) is set. If `ICANON` is **not** set, all characters are processed as regular characters and returned to the caller; only the `VMIN` and `VTIME` parameters are meaningful in the non-canonical processing mode.

The VEOL character can be used to signal end of line (and thus end of input in the canonical mode) in addition to the normal **RET** key. In the non-canonical mode, input ends as soon as at least VMIN characters are received.

Note that the values of VMIN and VTIME are currently ignored; termios functions always work as if VMIN were 1 and VTIME were zero. Other parameters are supported (for console devices only), except that VSTOP and VSTART characters are not inserted to the input, but otherwise produce no effect.

The c_cflag member of struct termios describes the hardware terminal control, as follows:

{Symbol {If set, send two stop bits

Note that since the DOS terminal doesn't use asynchronous ports, the above parameters are always ignored by the implementation. The default value of c_cflag is (CS8|CREAD|CLOCAL).

The c_lflag member of struct termios defines the local modes that control the terminal functions:

{Symbol {Canonical input (erase and kill processing)

The default value of c_lflag is (ISIG|ICANON|ECHO|IEXTEN|ECHOE|ECHOKE|ECHOCTL).

The c_iflag member of struct termios describes the input control:

{Symbol {Map upper-case to lower-case on input

The default value of c_iflag is (BRKINT|ICRNL|IMAXBEL).

The c_oflag member of struct termios specifies the output handling:

{Symbol {Map lower case to upper on output

Note that if the OPOST bit is not set, all the other flags are ignored and the characters are output verbatim. The default value of c_oflag is (OPOST|ONLCR|ONOEOT).

The c_ispeed and c_ospeed members specify, respectively, the input and output baudrate of the terminal. They are set by default to 9600 baud, but the value is always ignored by this implementation, since no asynchronous ports are used.

textattr

Syntax

```
#include <conio.h>

void textattr(int _attr);
```

Description

Sets the attribute used for future writes to the screen:

```
---- XXXX = foreground color
-XXX ---- = background color
X--- ---- = 1=blink 0=steady
```

The include file <conio.h> contains an enum COLORS that define the various values that can be used for these bitfields; light colors can only be used for the foreground.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
/* blinking white on blue */
textattr(BLINK | (BLUE << 4) | WHITE);
```

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

textbackground

Syntax

```
#include <conio.h>

void textbackground(int _color);
```

Description

Sets just the background of the text attribute. See textattr.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

textcolor

Syntax

```
#include <conio.h>

void textcolor(int _color);
```

Description

Sets just the foreground of the text attribute. See textattr.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

textmode

Syntax

```
#include <conio.h>

void textmode(int _mode);
```

Description

Sets the text mode of the screen. _mode is one of the following:

`LASTMODE`

The text mode which was in effect *before* the last call to `textmode()`.

`BW40`
: 40-column black and white (on a color screen)

`C40`
: 40-color color.

`BW80`
: 80-column black and white (on a color screen)

`C80`
: 80-column color

`MONO`
: The monochrome monitor

`C4350`
: 80-column, 43- (on EGAs) or 50-row (on VGAs) color

See _set_screen_lines, for a more versatile method of setting text screen dimensions.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

time

Syntax

```
#include <time.h>

time_t time(time_t *t);
```

Description

If t is not `NULL`, the current time is stored in `*t`.

Return Value

The current time is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
printf("Time is %d\n", time(0));
```

times

Syntax

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Description

This function returns the number of clock ticks used by the current process and all of its children until the moment of call. The number of tics per second is `CLOCKS_PER_SEC`, defined on time.h.

This is the structure in which `times` returns its info:

```
struct tms {
clock_t tms_cstime;
clock_t tms_cutime;
clock_t tms_stime;
clock_t tms_utime;
};
```

Currently, the elapsed time of the running program is returned in the `tms_utime` field, and all other fields return as zero.

Return Value

The number of elapsed tics since the program started.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
printf("We used %d seconds of elapsed time\n",
times(&buf)/CLOCKS_PER_SEC);
```

tmpfile

Syntax

```
#include <stdio.h>

FILE *tmpfile(void);
```

Description

This function opens a temporary file. It will automatically be removed if the file is closed or when the program exits. The name of the file is generated by the same algorithm as described under tmpnam() (See tmpnam).

Return Value

A newly opened file.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
FILE *tmp = tmpfile();
```

tmpnam

Syntax

```
#include <stdio.h>

char *tmpnam(char *s);
```

Description

This function generates a string that is a valid file name and that is not the same as the name of an existing file. A different string is guaranteed to be produced each time it is called, up to `TMP_MAX` times (`TMP_MAX` is defined on stdio.h). If `tmpnam` is called more than TMP_MAX times, the behavior is implementation-dependent (ours just wraps around and tries to reuse the same file names from the beginning).

This function examines the environment to determine the directory in which the temporary file will be opened. It looks for one of the variables "TMPDIR", "TEMP" and "TMP", in that order. The first one which is found in the environment will be used on the assumption that it points to a directory. If neither of the above variables is defined, tmpnam defaults to the "c:/" directory (which under MS-DOS might mean that it fails to generate TMP_MAX unique names, because DOS root directories cannot grow beyond certain limits).

Return Value

If s is a null pointer, tmpnam leaves its result in an internal static buffer and returns a pointer to that buffer. If s is not a null pointer, it is assumed to point to an array of at least L_tmpnam characters, and tmpnam writes its result in that array and returns a pointer to it as its value.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char buf[L_tmpnam];
char *s = tmpnam(buf);
```

toascii

Syntax

```
#include <ctype.h>

int toascii(int c);
```

Description

This function strips the high bit of c, forcing it to be an ASCII character.

Return Value

The ASCII character.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
for (i=0; buf[i]; i++)
buf[i] = toascii(buf[i]);
```

tolower

Syntax

```
#include <ctype.h>

int tolower(int c);
```

Description

This function returns c, converting it to lower case if it is upper case. See toupper.

Return Value

The lower case letter.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
for (i=0; buf[i]; i++)
buf[i] = tolower(buf[i]);
```

toupper

Syntax

```
#include <ctype.h>

int toupper(int c);
```

Description

This function returns c, converting it to upper case if it is lower case. See tolower.

Return Value

The upper case letter.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
for (i=0; buf[i]; i++)
buf[i] = toupper(buf[i]);
```

_truename

Syntax

```
#include <sys/stat.h>

char * _truename(const char *path, char *true_path);
```

Description

Given a path of a file, returns in true_path its canonicalized pathname, with all letters uppercased, default drive and directory made explicit, forward slashes converted to backslashes, asterisks converted to appropriate number of of question marks, file and directory names truncated to 8.3 if necessary, "." and ".." resolved, extra slashes (but the last, if present) removed, SUBSTed, JOINed and ASSIGNed drives resolved. Character devices return as "X:/DEVNAME" (note the forward slash!), where X is the CURRENT drive and DEVNAME is the device name (e.g. CON). This is exactly what DOS TRUENAME command does. See Ralph Brown's Interrupt List for more details.

The named path doesn't have to exist, but the drive, if given as part of it, should be a legal DOS drive, as this function hits the disk.

The function will fail if given a path which (1) is an empty string; or (2) contains only the drive letter (e.g. "c:"); or (3) has leading whitespace. It will also fail if it couldn't allocate memory required for its communication with DOS or for true_path (see below).

`_truename` may not return what you expect for files that don't exist. For instance, if the current directory was entered using a short filename (`c:\thisis~1` instead of `c:\thisisalongname`, say), then the truename of an existing file will be a long filename, but the truename of an non-existing file will be a short filename. This can cause problems when comparing filenames. Use `_truename_sfn` (See _truename_sfn) instead in this case.

Upon success, the function will place the result in true_path, if that's non-NULL; the buffer should be large enough to contain the largest possible pathname (PATH_MAX characters). If true_path is a NULL pointer, the space to hold the result will be allocated by calling `malloc` (See malloc); it is up to the caller to release the buffer by calling `free` (See free).

Return Value

The function returns the pointer to the result. In case of any failure, a NULL pointer is returned, and `errno` is set.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
fprintf(stderr,
"True name of %s is %s\n", path, _truename(path, (char *)0));
```

_truename_sfn

Syntax

```
#include <sys/stat.h>

char * _truename_sfn(const char *path, char *true_path);
```

Description

`_truename_sfn` is like `_truename`, except that it always returns a short filename. See the documentation for `_truename` for more details (See _truename).

Return Value

The function returns the pointer to the result. In case of any failure, a NULL pointer is returned, and `errno` is set.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
fprintf(stderr,
"True short name of %s is %s\n", path,
_truename_sfn(path, (char *)0));
```

truncate

Syntax

```
#include <unistd.h>

int truncate(const char *file, off_t size);
```

Description

This function truncates file to size bytes.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
truncate("/tmp/data.txt", 400);
```

ttyname

Syntax

```
#include <unistd.h>

char *ttyname(int file);
```

Description

Gives the name of the terminal associated with file.

Return Value

Returns "con" if file is a device, else `NULL`.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
char *tty = ttyname(0);
```

tzset

Syntax

```
#include <time.h>

extern char *tzname[2];
void tzset(void);
```

Description

This function initializes the global variable `tzname` according to environment variable `TZ`. After the call, `tzname` holds the specifications for the time zone for the standard and daylight-saving times.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

tzsetwall

Syntax

```
#include <time.h>

void tzsetwall(void);
```

Description

This function sets up the time conversion information used by `localtime` (See localtime) so that `localtime` returns the best available approximation of the local wall clock time.

Return Value

None.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

uclock

Syntax

```
#include <time.h>

uclock_t uclock(void);
```

Description

This function returns the number of uclock ticks since an arbitrary time, actually, since the first call to `uclock`, which itself returns zero. The number of tics per second is `UCLOCKS_PER_SEC` (declared in the `time.h` header file.

`uclock` is provided for very high-resulution timing. It is currently accurate to better than 1 microsecond (actually about 840 nanoseconds). You cannot time across two midnights with this implementation, giving a maximum useful period of 48 hours and an effective limit of 24 hours. Casting to a 32-bit integer limits its usefulness to about an hour before 32 bits will wrap.

Note that `printf` will only print a value of type `uclock_t` correctly if you use the format specifier for `long`

`long` data, `%lld`, because `uclock_t` is a 64-bit integer. See printf.

Also note that `uclock` reprograms the interval timer in your PC to act as a rate generator rather than a square wave generator. I've had no problems running in this mode all the time, but if you notice strange things happening with the clock (losing time) after using `uclock`, check to see if this is the cause of the problem.

Windows 3.X doesn't allow to reprogram the timer, so the values returned by `uclock` there are incorrect. DOS and Windows 9X don't have this problem.

Windows NT, 2000 and XP attempt to use the `rdtsc` feature of newer CPUs instead of the interval timer, because the timer tick and interval timer are not coordinated. During calibration the `SIGILL` signal handler is replaced to protect against systems which do not support or allow `rdtsc`. If `rdtsc` is available, uclock will keep the upper bits of the returned value consistent with the bios tick counter by re-calibration if needed. If `rdtsc` is not available, these systems fall back to interval timer usage, which may show an absolute error of 65536 uclock ticks in the values and not be monotonically increasing.

Return Value

The number of tics.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
printf("%lld ticks have elapsed\n", (long long)(uclock()));
printf("%f second have elapsed\n",
((double)uclock()/UCLOCKS_PER_SEC));
```

umask

Syntax

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Description

This function does nothing. It exists to assist porting.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

uname

Syntax

```
#include <sys/utsname.h>

int uname(struct utsname *u);
```

Description

Fills in the structure with information about the system.

```
struct utsname {
char machine[9];
char nodename[32];
char release[9];
char sysname[9];
char version[9];
};
```

`machine`
: The CPU family type: `"i386"`, `"i486"`, `"i586"` (Pentium), `"i686"` (PentiumPro, Pentium II, Pentium III), or `"i786"` (Pentium 4).

`nodename`
: The name of your PC (if networking is installed), else `"pc"`.

`release`
: The major version number of DOS. For example, DOS 1.23 would return `"1"` here.

`sysname`
: The flavor of the OS.

`version`
: The minor version number of DOS. For example, DOS 1.23 would return `"23"` here.

Return Value

Zero on success, else nonzero.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

ungetc

Syntax

```
#include <stdio.h>

int ungetc(int c, FILE *file);
```

Description

This function pushes c back into the file. You can only push back one character at a time.

Return Value

The pushed-back character, or `EOF` on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int q;
while (q = getc(stdin) != 'q');
ungetc(q);
```

ungetch

Syntax

```
#include <conio.h>

int ungetch(int);
```

Description

Puts a character back, so that `getch` (See getch) will return it instead of actually reading the console.

Return Value

The charater is returned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

unlink

Syntax

```
#include <unistd.h>

int unlink(const char *file);
```

Description

This function removes a file from the file system.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992 (partial); 1003.1-2001 (partial) (see note 1)

Notes:

1. The POSIX specification requires this removal to be delayed until the file is no longer open. Due to problems with the underlying operating systems, this implementation of `unlink` does not fully comply with the specs; if the file you want to unlink is open, you're asking for trouble --- how much trouble depends on the underlying OS. On Windows NT (and possibly on Windows 2000 as well), you get the behaviour POSIX expects. On Windows 9x and Windows ME (and possibly Windows XP as well), the removal will simply fail (`errno` gets set to `EACCES`). On DOS, removing an open file could lead to filesystem corruption if the removed file is written to before it's closed.

Example

```
unlink("data.txt");
```

unlock

Syntax

```
#include <io.h>

int unlock(int fd, long offset, long length);
```

Description

Unlocks a region previously locked by `lock`.

See lock.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

unlock64

Syntax

```
#include <io.h>

int unlock64(int fd, long long offset, long long length);
```

Description

Unlocks a region previously locked by `lock64`.

Arguments offset and length must be of type `long long`, thus enabling you to unlock with offsets and lengths as large as ~2^63 (FAT16 limits this to ~2^31; FAT32 limits this to 2^32-2).

See lock64.

Return Value

Zero if successful, nonzero if not.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

unsetenv

Syntax

```
#include <stdlib.h>

int unsetenv(const char *name);
```

Description

This function removes the environment variable name from the environment. This will update the list of pointers to which the environ variable points. If the specified variable does not exist in the environment, the environment is not modified and this function is considered to have been sucessfully completed.

Return Value

If name is `NULL`, points to an empty string, or points to a string containing a `=`, this function returns -1 and sets `errno` to `EINVAL`; otherwise it returns 0.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001 (see note 1)

Notes:

1. This function is new to the Posix 1003.1-200x draft

_use_lfn

Syntax

```
#include <fcntl.h>
char _use_lfn(const char *path);
```

Description

The `_use_lfn` function returns a nonzero value if the low level libc routines will use the Long File Name (LFN) functions provided with Windows 9x (and other advanced filesystems), when accessing files and directories on the same filesystem as path. path may be any legal pathname; however, the function only needs the name of the root directory on the particular drive in question. If path is a `NULL` pointer, the function assumes that all the filesystems support (or do not support) LFN in the same manner, and returns the info pertinent to the last filesystem that was queried; this usually makes the call faster. Note that on Windows 95 you don't need to distinguish between different drives: they all support LFN API. If path does not specify the drive explicitly, the current drive is used.

The header `fcntl.h` defines a macro `_USE_LFN`; applications should use this macro instead of calling `_use_lfn` directly. That is so this routine could be replaced with one which always returns 0 to disable using long file names. Calling `_USE_LFN` also makes the code more portable to other operating systems, where the macro can be redefined to whatever is appropriate for that environment (e.g., it should be a constant 1 on Unix systems and constant 0 for environments that don't support LFN API, like some other MSDOS compilers). Currently, `_USE_LFN` assumes that LFN API does *not* depend on a drive.

Long file names can also be disabled by setting the flag `_CRT0_FLAG_NO_LFN` in `_crt0_startup_flags` for an image which should not allow use of long file names. Long names can be suppressed at runtime on a global basis by setting the environment variable `LFN` to `N`, i.e. `SET LFN=N`. This might be needed if a distribution expected the truncation of long file names to 8.3 format to work. For example, if a C source routine included the file exception.h (9 letters) and the file was unzipped as exceptio.h, then GCC would not find the file unless you set `LFN=n`. The environment variable can be set in the `DJGPP.ENV` file to always disable LFN support on any system, or can be set in the DOS environment for a short term (single project) basis. If you dual boot a system between Windows 95 and DOS, you probably should set `LFN=n` in your `DJGPP.ENV` file, since long file names would not be visible under DOS, and working with the short names under DOS will damage the long names when returning to Windows 95.

Return Value

If LFN APIs are supported and should be used, it returns 1, else 0.

Note that if the `_CRT0_FLAG_NO_LFN` bit is set, or `LFN` is set to `N` or `n` in the environment, both `_use_lfn` and `_USE_LFN` will always return 0 without querying the filesystem. You can reset the `_CRT0_FLAG_NO_LFN` bit at runtime to force filesystem to be queried.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
#include <fcntl.h>
#include <sys/stat.h>

int fd = creat (_USE_LFN ? "MyCurrentJobFile.Text" : "currjob.txt",
S_IRUSR | S_IWUSR);
```

usleep

Syntax

```
#include <unistd.h>

unsigned usleep(unsigned usec);
```

Description

This function pauses the program for usec microseconds. Note that, since `usleep` calls `clock` internally, and the latter has a 55-msec granularity, any argument less than 55msec will result in a pause of random length between 0 and 55 msec. Any argument less than 11msec (more precisely, less than 11264 microseconds), will always result in zero-length pause (because `clock` multiplies the timer count by 5). See clock.

Return Value

The number of unslept microseconds (i.e. zero).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
usleep(500000);
```

utime

Syntax

```
#include <utime.h>

int utime(const char *file, const struct utimbuf *time);
```

Description

This function sets the modification timestamp on the file. The new time is stored in this structure:

```
struct utimbuf
```

```
{

time_t actime; /* access time (unused on FAT filesystems) */
time_t modtime; /* modification time */
};
```

Note that, as under DOS a file only has a single timestamp, the `actime` field of `struct utimbuf` is ignored by this function, and only `modtime` field is used. On filesystems which support long filenames, both fields are used and both access and modification times are set.

If file is a directory, the function always fails, except on Windows 2000 and Windows XP, because other systems don't allow changing the time stamp of a directory.

Return Value

Zero for success, nonzero for failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

```
struct utimbuf t;
t.modtime = time(0);
utime("data.txt", &t);
```

utimes

Syntax

```
#include <sys/time.h>

int utimes(const char *file, struct timeval tvp[2]);
```

Description

This function sets the file access time as specified by `tvp[0]`, and its modification time as specified by `tvp[1]`. `struct timeval` is defined as follows:

```
struct timeval {
time_t tv_sec;
long tv_usec;
};
```

Note that DOS and Windows maintain the file times with 2-second granularity. Therefore, the `tv_usec` member of the argument is always ignored, and the underlying filesystem truncates (or sometimes rounds) the actual file time stamp to the multiple of 2 seconds.

On plain DOS, only one file time is maintained, which is arbitrarily taken from `tvp[1].tv_sec`. On Windows 9X, both times are used, but note that most versions of Windows only use the date part and ignore the time.

Due to limitations of DOS and Windows, you cannot set times of directories.

Return Value

Zero on success, nonzero on failure.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
time_t now;
struct timeval tvp[2];
time(&now);
tvp[1].tv_sec = now + 100;
utimes("foo.dat", tvp);
```

v2loadimage

Syntax

```
#include <debug/v2load.h>

int v2loadimage (const char *program, const char *cmdline,
jmp_buf load_state);
```

Description

This function loads an executable image of a DJGPP v2.x program and prepares it for debugging. program should point to the file name of the executable program. `v2loadimage` does **not** search the `PATH` and does **not** try any executable extensions, so program should point to a fully-qualified path, complete with the drive, directory, and file-name extension; otherwise the call will fail.

cmdline should point to the command-line arguments to be passed to the program. A command line up to 126 characters long can be formatted exactly like the command tail DOS passes to programs: the first byte gives the length of the command tail, the tail itself begins with the second byte, and the tail is terminated by a CR character (decimal code 13); the length byte does not include the CR. Longer command lines require a different format: the first byte is 255, the command-line starting with the second byte which is terminated by a NUL character (decimal code 0). Regardless of the method used, the command-line arguments should look as if they were to be passed to the library function `system`. In particular, all special characters like wildcards and whitespace should be quoted as if they were typed at the DOS prompt.

After the function loads the image and sets up the necessary memory segments for it to be able to run, it sets load_state so that it can be used to `longjmp` to the debuggee's entry point. This information is typically used by `run_child` (See run_child).

Return Value

Zero in case of success, non-zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
cmdline = (char *) alloca (strlen (args) + 4);
cmdline[0] = strlen (args);
strcpy (cmdline + 1, args);
cmdline[strlen (args) + 1] = 13;

if (v2loadimage (exec_file, cmdline, start_state))
{
printf ("Load failed for image %s\n", exec_file);
exit (1);
}

edi_init (start_state);
```

valloc

Syntax

```
#include <stdlib.h>

void *valloc(size_t size);
```

Description

This function is just like `malloc` (See malloc) except the returned pointer is a multiple of the CPU page size which is 4096 bytes.

Return Value

A pointer to a newly allocated block of memory.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *page = valloc(getpagesize());
```

varargs

Syntax

```
#include <stdarg.h>

void va_start(va_list ap, LAST_REQUIRED_ARG);
TYPE va_arg(va_list ap, TYPE);
void va_end(va_list ap);
```

Description

Used to write functions taking a variable number of arguments. Note that these are actually macros, and not functions. You must prototype the function with '…' in its argument list. Then, you do the following:

- Create a variable of type `va_list`.
- Initialize it by calling `va_start` with it and the name of the last required (i.e. non-variable) argument.
- Retrieve the arguments by calling `va_arg` with the `va_list` variable and the type of the argument. As another alternative, you can pass the `va_list` to another function, which may then use `va_arg` to get at the arguments. `vprintf` is an example of this.
- Call `va_end` to destroy the `va_list`.

Be aware that your function must have some way to determine the number and types of the arguments. Usually this comes from one of the required arguments. Some popular ways are to pass a count, or to pass some special value (like `NULL`) at the end.

Also, the variable arguments will be promoted according to standard C promotion rules. Arguments of type `char` and `short` will be promoted to `int`, and you should retrieve them as such. Those of type `float` will be promoted to `double`.

Return Value

`va_arg` returns the argument it fetched, the other macros return nothing.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int find_the_sum(int count, …)
{

va_list ap;
int i;
int total = 0;
va_start(ap, count);
for (i = 0; i < count; i++)
total += va_arg(ap, int);
va_end(ap);
return total;
}


int other_function(void)
```

```
{

return find_the_sum(6, 1, 2, 3, 4, 5, 6);
}
```

vfork

Syntax

```
#include <unistd.h>

pid_t vfork(void);
```

Description

This function always returns -1 and sets 'errno' to ENOMEM, as MS-DOS does not support multiple processes. It exists only to assist in porting Unix programs.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

vfprintf

Syntax

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

int vfprintf(FILE *file, const char *format, va_list arguments);
```

Description

Sends formatted output from the arguments to the file. See printf.

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
void my_errmsg(char *format, ...)
{

va_list arg;

va_start(arg, format);
fprintf(stderr, "my_errmsg: ");
vfprintf(stderr, format, arg);
va_end(arg);
}
```

vfscanf

Syntax

```
#include <stdio.h>

int vfscanf(FILE *file, const char *format, va_list arguments);
```

Description

This function scans formatted text from file and stores it in the variables pointed to by the arguments. See scanf.

Return Value

The number of items successfully scanned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

vprintf

Syntax

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

int vprintf(const char *format, va_list arguments);
```

Description

Sends formatted output from the arguments to `stdout`. See printf. See vfprintf.

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

vscanf

Syntax

```
#include <stdio.h>

int vscanf(const char *format, va_list arguments);
```

Description

This function scans formatted text from `stdin` and stores it in the variables pointed to by the arguments. See scanf. See vfscanf.

Return Value

The number of items successfully scanned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

vsnprintf

Syntax

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

int vsnprintf (char *buffer, size_t n, const char *format,
va_list ap);
```

Description

This function works similarly to `vsprintf()` (See vsprintf), but the size n of the buffer is also taken into account. This function will write n - 1 characters. The nth character is used for the terminating nul. If n is zero, buffer is not touched.

Return Value

The number of characters that would have been written (excluding the trailing nul) is returned; otherwise -1 is returned to flag encoding or buffer space errors.

The maximum accepted value of n is `INT_MAX`. `INT_MAX` is defined in `<limits.h>`. -1 is returned and `errno` is set to EFBIG, if n is greater than this limit.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 (see note 1)

Notes:

1. The buffer size limit is imposed by DJGPP. Other systems may not have this limitation.

vsprintf

Syntax

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

int vsprintf(char *buffer, const char *format, va_list arguments);
```

Description

Sends formatted output from the arguments to the buffer. See printf. See vfprintf.

To avoid buffer overruns, it is safer to use `vsnprintf()` (See vsnprintf).

Return Value

The number of characters written.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

vsscanf

Syntax

```
#include <stdio.h>

int vsscanf(const char *string, const char *format, va_list arguments);
```

Description

This function scans formatted text from the string and stores it in the variables pointed to by the arguments. See scanf. See vfscanf.

Return Value

The number of items successfully scanned.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

wait

Syntax

```
#include <sys/wait.h>

pid_t pid = wait(int *status);
```

Description

This function causes its caller to delay its execution until a signal is received or one of its child processes terminates. If any child has terminated, return is immediate, returning the process ID and its exit status, if that's available. If no children processes were called since the last call, then -1 is returned and `errno` is set.

Return Value

If successful, this function returns the exit status of the child. If there is an error, these functions return -1 and set `errno` to indicate the error type.

Bugs

Currently, this function always fails.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

waitpid

Syntax

```
#include <sys/wait.h>

pid_t pid = waitpid((pid_t pid, int *status, int options);
```

Description

Currently, this function always fails. A -1 is returned and `errno` is set to indicate there are no children.

Return Value

If successful, this function returns the exit status of the child. If there is an error, these functions return -1 and set `errno` to indicate the error type.

Bugs

Currently, this function always fails.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

wcstombs

Syntax

```
#include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
```

Description

Converts a wide character string to a multibyte string. At most n characters are stored.

Return Value

The number of characters stored.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
int len = wcstombs(buf, wstring, sizeof(buf));
```

wctomb

Syntax

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

Description

Convert a wide character to a multibyte character. The string s must be at least `MB_LEN_MAX` bytes long.

Return Value

The number of characters stored.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx C89; C99 1003.2-1992; 1003.1-2001

Example

```
char s[MB_LEN_MAX];
int mlen = wctomb(s, wc);
```

wherex

Syntax

```
#include <conio.h>

int wherex(void);
```

Return Value

The column the cursor is on. The leftmost column is 1.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

wherey

Syntax

```
#include <conio.h>

int wherey(void);
```

Return Value

The row the cursor is on. The topmost row is 1.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

wild

Syntax

```
#include <debug/wild.h>

int wild (char *pattern, char *string);
```

Description

This function matches a string pointed to by string against a pattern pointed to by pattern. pattern may include wildcard characters `?` and `*`, meaning, respectively, any single character and any string of characters. The function returns non-zero if the string matches the pattern, zero otherwise.

This function is meant to be used for simple matching of patterns, such as if a debugger needs to allow specification of symbols using wildcards.

Return Value

The function returns non-zero if the string matches the pattern, zero otherwise.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

window

Syntax

```
#include <conio.h>

void window(int _left, int _top, int _right, int _bottom);
```

Description

Specifies the window on the screen to be used for future output requests. The upper left corner of the physical screen is (1,1).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Implementation Note

It's not safe to call this function inside static constructors, because conio needs to be initialized, and its initialization is done by a static constructor. Since you don't have any control on the order in which static constructors are called (it's entirely up to the linker), you could have problems.

If you can detect the situation when one of the conio functions is called for the very first time since program start, you could work around this problem by calling the `gppconio_init` function manually (this is the function called by a static constructor).

write

Syntax

```
#include <unistd.h>

int write(int file, const void *buffer, size_t count);
```

Description

This function writes count bytes from buffer to file. It returns the number of bytes actually written. It will return zero or a number less than count if the disk is full, and may return less than count even under valid conditions.

Note that if file is a text file, `write` may write more bytes than it reports.

If count is zero, the function does nothing and returns zero. Use `_write` if you want to actually ask DOS to write zero bytes.

The precise behavior of `write` when the target filesystem is full is somewhat troublesome, because DOS doesn't fail the underlying system call. If your application needs to rely on `errno` being set to `ENOSPC` in such cases, you need to invoke `write` as shown in the example below. In a nutshell, the trick is to call `write` one more time after it returns a value smaller than the count parameter; then it will *always* set `errno` if the disk is full.

Return Value

The number of bytes written, zero at EOF, or -1 on error.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No 1003.2-1992; 1003.1-2001

Example

This example shows how to call `write` in a way which ensures that `errno` will be set to `ENOSPC` if the target filesystem is or becomes full:

```
char *buf_ptr; /* the buffer to write */
size_t buf_len; /* the number of bytes to write */
int desc; /* the file descriptor to write to */

while (buf_len > 0)
{
int written = write (desc, buf_ptr, buf_len);
if (written <= 0)
break;

buf_ptr += written;
buf_len -= written;
}
```

_write

Syntax

```
#include <io.h>

ssize_t _write(int fildes, const void *buf, size_t nbyte);
```

Description

This is a direct connection to the MS-DOS write function call, int 0x21, %ah = 0x40. No conversion is done on the data; it is written as raw binary data. This function can be hooked by the File-system extensions, see See File System Extensions. If you don't want this, you should use `_dos_write` instead, see See _dos_write.

Return Value

The number of bytes written, or -1 (and `errno` set) in case of failure.

Note that DOS doesn't return an error indication when the target disk is full; therefore if the disk fills up while the data is written, `_write` does **not** return -1, it returns the number of bytes it succeeded to write. If you need to detect the disk full condition reliably, call `_write` again to try to write the rest of the data. This will cause DOS to return zero as the number of written bytes, and *then* `_write` will return -1 and set `errno` to `ENOSPC`. The example below shows one way of doing this.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

This example shows how to call `_write` in a way which ensures that `errno` will be set to `ENOSPC` if the target filesystem is or becomes full:

```
char *buf_ptr; /* the buffer to write */
size_t buf_len; /* the number of bytes to write */
int desc; /* the file descriptor to write to */

while (buf_len > 0)
{
int written = _write (desc, buf_ptr, buf_len);
```

```
if (written <= 0)
break;

buf_ptr += written;
buf_len -= written;
}
```

write_child

Syntax

```
#include <debug/dbgcom.h>

void write_child (unsigned child_addr, void *buf, unsigned len);
```

Description

This function transfers len bytes from the buffer pointed to by buf in the debugger's data segment to the memory of the debugged process starting at the address child_addr. It is used primarily to insert a breakpoint instruction into the debugged process (to trigger a trap when the debuggee's code gets to that point). The companion function `read_child` (See read_child) is usually called before `write_child` to save the original code overwritten by the breakpoint instruction.

Return Value

The function return zero if it has successfully transferred the data, non-zero otherwise (e.g., if the address in child_addr is outside the limits of the debuggee's code segment.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

write_sel_addr

Syntax

```
#include <debug/dbgcom.h>

void write_sel_addr (unsigned sel, unsigned offset,
void *buf, unsigned len);
```

Description

This function transfers len bytes from the buffer pointed to by buf in the data segment whose selector is sel, at offset offset. The companion function `read_sel_addr` (See read_sel_addr) is usually called before `write_sel_addr` to save the original contents, if needed.

Return Value

The function return zero if it has successfully transferred the data, non-zero otherwise (e.g., if the address in offset is outside the limits of the sels segment).

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

xfree

Syntax

```
#include <stdlib.h>

void xfree(void *ptr);
```

Description

Frees memory allocated by `xmalloc` (See xmalloc). This function guarantees that a NULL pointer is handled gracefully.

Note that, currently, the header `stdlib.h` does **not** declare a prototype for `xfree`, because many programs declare its prototype in different and conflicting ways. If you use `xfree` in your own code, you might need to provide your own prototype explicitly.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
void *f = xmalloc(100);
xfree(f);
```

xmalloc

Syntax

```
#include <stdlib.h>

void *xmalloc(size_t size);
```

Description

This function is just like `malloc` (See malloc), except that if there is no more memory, it prints an error message and exits.

Note that, currently, the header `stdlib.h` does **not** declare a prototype for `xmalloc`, because many programs declare its prototype in different and conflicting ways. If you use `xmalloc` in your own code, you might need to provide your own prototype explicitly.

Return Value

A pointer to the newly allocated memory.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *f = xmalloc(100);
```

xrealloc

Syntax

```
#include <stdlib.h>

void *xrealloc(void *ptr, size_t size);
```

Description

This function is just like `realloc` (See realloc), except that if there is no more memory, it prints an error message and exits. It can also properly handle ptr being `NULL`.

Note that, currently, the header `stdlib.h` does **not** declare a prototype for `xrealloc`, because many programs declare its prototype in different and conflicting ways. If you use `xrealloc` in your own code, you might need to provide your own prototype explicitly.

Return Value

A pointer to a possibly new block.

Portability

{ANSI/ISO C {xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx No No

Example

```
char *buf;
buf = (char *)xrealloc(buf, new_size);
```

Unimplemented Functions

The DJGPP standard C library is ANSI- and POSIX-compliant, and provides many additional functions for compatibility with Unix/Linux systems. However, some of the functions needed for this compatibility are very hard or impossible to implement using DOS facilities.

Therefore, a small number of library functions are really just stubs: they are provided because POSIX requires them to be present in a compliant library, or because they are widely available on Unix systems, but they either always fail, or handle only the trivial cases and fail for all the others. An example of the former behavior is the function `fork`: it always returns a failure indication; an example of the latter behavior is the function `mknode`: it handles the cases of regular files and existing character devices, but fails for all other file types.

This chapter lists all such functions. This list is here for the benefit of programmers who write portable programs or port Unix packages to DJGPP.

Each function below is labeled as either ‘‘always fails’’ or ‘‘trivial’’, depending on which of the two classes described above it belongs to. An additional class, labeled ‘‘no-op’’, includes functions which pretend to succeed, but have no real effect, since the underlying functionality is either always available or always ignored.

Function Index

Variable Index

Data Type Index

Concept Index