_fmode

Default buffered I/O mode

Class: Lattice

Category: Stream I/O

SYNOPSIS

```
extern int _fmode;
```

DESCRIPTION

This external integer is used by the fopen function to determine the translation mode to use when the programmer does not specify a mode in the fopen call. For GEMDOS it is set to 0, which specifies translated mode. If the default is to be binary mode the variable should be set to the value O_RAW defined in fcntl.h.

SEE

fopen

fmode

Change mode of buffered file

Class: Lattice

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

fmode(fp,mode);

FILE *fp;          file pointer
int mode;          0 => mode A
                   1 => mode B
```

DESCRIPTION

This function is used to change the translation mode of a file that has been opened via fopen, freopen, or fdopen.

In mode A, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In mode B, all data is transferred with no changes.

The file pointer is not checked for validity.

SEE

fopen, freopen, fdopen

fopen

Open a buffered file

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

fp = fopen(name, mode);

FILE *fp;           file pointer
const char *name;   file name
const char *mode;   access mode
```

DESCRIPTION

This function opens a file for buffered access. The name string can be any valid file name and may include a device code and directory path. The mode string indicates how the file is to be processed, as follows:

Mode	Create	Truncate	Read	Write	Append	Translate
"r"	No	No	Yes	No	No	Default
"w"	Yes	Yes	No	Yes	No	Default
"a"	Yes	No	No	No	Yes	Default
"r+"	No	No	Yes	Yes	No	Default
"w+"	Yes	Yes	Yes	Yes	No	Default
"a+"	Yes	No	Yes	No	Yes	Default
"ra"	No	No	Yes	No	No	ModeA
"wa"	Yes	Yes	No	Yes	No	ModeA
"aa"	Yes	No	No	No	Yes	ModeA
"ra+"	No	No	Yes	Yes	No	ModeA
"wa+"	Yes	Yes	Yes	Yes	No	ModeA
"aa+"	Yes	No	Yes	No	Yes	ModeA
"rb"	No	No	Yes	No	No	ModeB
"wb"	Yes	Yes	No	Yes	No	ModeB
"ab"	Yes	No	No	No	Yes	ModeB
"rb+"	No	No	Yes	Yes	No	ModeB
"wb+"	Yes	Yes	Yes	Yes	No	ModeB
"ab+"	Yes	No	Yes	No	Yes	ModeB

The following comments explain the columns in the previous table:

	Yes	No
Create	The file will be created if it does not already exist.	The function will fail if the file does not already exist.
Truncate	If the file exists, it will be truncated (i.e. marked as empty).	If the file exists, its current contents will not be disturbed.
Read	The file can be read via functions such as fread and fgetc. Also, fseek can be used to position the file before reading.	The file cannot be read.
Write	The file can be written via functions such as fwrite and fputc. Also, fseek can be used to positon the file before writing.	The file cannot be written, but see Append below.
Append	The file can be written, but it is automatically positioned to the current end-of-file before each write operation. This effectively prevents existing data from being changed.	Automatic positioning to the end-of-file is not done before a write operation. Also, writes are not allowed unless Write is "Yes".

TRANSLATE - Default

The external integer _fmode is used to set mode A or mode B as follows:

```
if(_fmode & 0x8000)
  set mode B
else
  set mode A
```

TRANSLATE - Mode A

On a read operation, each carriage return character ('\r') is deleted. On a write operation, each line feed character ('\n') is expanded to a carriage return followed by a line feed.

TRANSLATE - Mode B

The data is unchanged as it is read or written.

If the file is successfully opened, the function returns a pointer to a "buffered I/O control block", which is defined in the header file stdio.h. Normally you will not need to access any information in the control block directly, but you should be very careful not to disturb the block accidentally. A common C programming error is to accidentally mutilate one of these control blocks, which can cause garbage to be written into a file.

RETURNS

If the operation is successful, the function returns a non-NULL file pointer. A NULL pointer is returned if the file cannot be opened. Consult errno and _OSERR for detailed error information.

When a file is opened for both reading and writing, you should call fseek or rewind when switching from reading to writing or vice-versa. It is not necessary to do this when you begin writing after reading up to the end of the file.

SEE

fclose, fdopen, fgetc, fgets, fputc, fputs, fread, freopen, fwrite

fopene

Perform fopen with environment search

Class: Lattice

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

fp = fopene(name,mode,path);

FILE *fp;           file pointer
const char *name;   file name
const char *mode;   buffered file access mode
char *path;         path return
```

DESCRIPTION

The fopene function is like fopen except that it performs an extended directory search for file names that cannot be found in the current directory. The directory searching algorithm is:

- Try the file name as specified. If successful, return the file pointer. Otherwise, if the name is absolute, indicate an error. An absolute name begins with a slash (/), a backslash (\), or has a colon (:) in the second character. If the name is relative, continue.
- Check if the file name has an extension. If so, convert the extension to upper case and look for an environment variable of that name. If the variable is found, it should consist of a list of alternate directories separated by semicolons (;) or commas (,). Append the file name to each directory name in turn, and retry the open operation. If successful, copy the directory name to the path argument, if that argument is not NULL, and then return the file pointer. If unsuccessful, continue.
- Find the environment variable named PATH and repeat the preceding step with those directory names. If unsuccessful, return an error indication.

RETURNS

If the operation is successful, the function returns a non-NULL file pointer. A NULL pointer is returned if the file cannot be opened. Consult errno and _OSERR for detailed error information.

SEE

fopen, open, opene

EXAMPLE

```
Assume that the following environment variables have
been set up:

PATH=c:\bin;c:\dos
C=source
```

Then if you attempt to open the file named "myprog.c", the fopene or opene function will try the following names, in this order:

```
myprog.c
source\myprog.c
c:\bin\myprog.c
c:\dos\myprog.c
```


fork

Create a child process

Class: Lattice

Category: Process Creation

SYNOPSIS

```
#include  <stdlib.h>

error =  forkl(prog,arg0,arg1,...,argn,NULL);
error =  forkv(prog,argv);

error =  forkle(prog,arg0,arg1,...,argn,NULL,envp);
error =  forkve(prog,argv,envp);

error =  forklp(prog,arg0,arg1,...,argn,NULL);
error =  forkvp(prog,argv);

error =  forklpe(prog,arg0,arg1,...,argn,NULL,envp);
error =  forkvpe(prog,argv,envp);

int  error;              error  code
const  char  *prog;      program  name
const  char  *arg0;      argument  #0
const  char  *arg1;      argument  #1
const  char  *argn;      argument  #n
const  char  *argv[];    argument  vector
const  char  *envp[];    environment  pointers

extern  int  _aecl;      Atari  extended  command  lines
                         flag
```

DESCRIPTION

These functions create a "child process" by loading a new program and passing control to it. When the child process completes, the current program (i.e. the "parent process") can obtain its completion code via the wait function.

When a child process is created under GEMDOS, the parent suspends execution until the child is finished.

You can specify the arguments for the child program in two ways. In the "list method," the function call includes a list of argument string pointers terminated by a NULL pointer. In the "vector method," the function call includes a single pointer to an array of argument string pointers, with the array being terminated by a NULL pointer. Following UNIX conventions, the first argument (i.e. arg0 or argv(0)) should be the program name and is normally the same as prog. The arguments are all passed to the child process using the Atari extended command line format, so that the number of arguments is limited only by memory. The arguments are also concatenated into a pseudo-command line, with a blank separating adjacent arguments, so that naïve children may obtain a command line. The maximum size of this line is 127 bytes under GEMDOS.

Note that the use of extended command lines may be disabled by setting the external variable _aecl to 0. This defaults to 1, i.e. on.

The forkl, forkle, forkv, and forkve functions look for the program file only in the current directory. The other functions make an extended search using the PATH environment variable. The search procedure is:

- Search the current directory. If the program name has no extension, first search for a file with a .PRG extension, then .TTP, .TOS and .APP. If any of these searches succeeds, use that file for execution. If all searches fail and this is the forkl, forkle, forkv, or forkve function, return an error code. Otherwise proceed to the next step.

- Find the PATH environment variable; if it does not exist, indicate failure. Otherwise, perform the search as above in each directory listed. If all searches fail, return an error code.

For the functions that end with an "e", the envp array specifies a new set of environment variables that will be passed to the new program. This array is similar to argv, in that it must contain one or more pointers to strings and must end with a NULL pointer. Furthermore, the environment strings must each have the form "name=value".

RETURNS

If the function call is successful, 0 is returned. If the specified program file cannot be found, a -1 return is made, and additional error information can be found in errno and _OSERR. Note that you must call the wait function in order to obtain the completion code from the child process.

SEE

Pexec, exit, wait

EXAMPLE

```
/*
 * This program prints the environment,
 * prompts for additional environment strings,
 * and then forks a copy of itself. This
 * continues until you run out of memory or
 * abort via CTRL C.
 *
 */
#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <errno.h>
#include  <dos.h>

extern  char  **environ;

int  main(void)
{
  int  x;
  char  *q,b[100];

  prenv();
  for  (;;)
  {
    printf("Type  env  string  (e.g.  xx=yy),or  ENTER\n");
    if(!gets(b))
      break;
    if(b[0]  !=  '\0')
    {
      q  =  strdup(b);
      if(!q)
      {
        printf("Out  of  memory\n");
        break;
      }
      if(putenv(q))
      {
        perror("putenv");
        break;
      }
    }
    else
      break;
  }

  if(x  =  forkl("fork","fork",NULL))
    printf("\nFORK  ERROR  %d  errno=%d  _OSERR=%d\n",
      x,errno,_OSERR);
  printf("DONE  %x\n",_OSERR);
}

void  prenv(void)
{
  char  **p;
  printf("\nENVIRONMENT...\n");
  for(p  =  environ;  *p;  p++)
    printf("%s\n",*p);
  printf("***DONE***\n\n");
}
```

_FPERR — Floating Point Error Code

Class: Lattice *Category: Errors*

SYNOPSIS

```
extern int _FPERR;
```

DESCRIPTION

This location will contain a non-zero value after any low-level floating point operation encounters an error. Low-level operations include addition, subtraction, multiplication, division, comparison, and conversion from one number representation to another (e.g. float to double).

The error codes and their corresponding symbols from math.h:

Symbol	Value	Meaning
FPEUND	1	Underflow
FPEOVF	2	Overflow
FPEDVZ	3	Divide by zero
FPENAN	4	Not a valid number
FPECOM	5	Not comparable

When the error occurs, the low-level operation passes the appropriate error code to _CXFERR, which must store the code in _FPERR. Note that _FPERR is never reset by any low-level operation.

SEE

_CXFERR

EXAMPLE

```
/*
 * This example performs uses the division operation
 * to stimulate floating point errors.
 */

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

int main(void)
{
  double a,b,c;
  extern int _FPERR;

  while(!feof(stdin))
  {
    printf("Enter divisor: ");
    if(scanf("%lf",&a) != 1)
      break;
    printf("Enter dividend: ");
    if(scanf("%lf",&b) != 1)
      break;
    _FPERR = 0;
    c = b / a;
    printf("_FPERR = %d\n",_FPERR);
    printf("%e / %e = %e\n\n",b,a,c);
  }
  return 0;
}
```

fprintf

Formatted print to a file

Class: ANSI

Category: Formatted I/O

SYNOPSIS

```
#include <stdio.h>

length = fprintf(fp,fmt,arg1,arg2,...);

int length;         number of characters generated
const char *fmt;    format string
FILE *fp;           file pointer

See printf for arg1, arg2, and so on.
```

DESCRIPTION

The printf group of functions generate a stream of ASCII characters by analysing the format string and performing various conversion operations on the remaining arguments. The fprintf form of printf sends the output stream to the file specified by fp.

See the description of the printf function for a complete discussion of the arguments and conversion specifications.

RETURNS

This function returns the number of output characters generated.

SEE

cprintf, lprintf, printf, sprintf, vfprintf, vprintf, vsprintf

fputc, fputchar

Put a character to a file/stdout

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

r = fputc(c,fp);    Put a character to a buffered file
r = fputchar(c);    Put a character to stdout

int r;              EOF or c
int c;              Character to be output
FILE *fp;           File pointer
```

DESCRIPTION

These functions put a single character to the specified file previously opened via fopen, freopen, or fdopen. The standard output file, stdout, is used for fputchar.

RETURNS

The output character is returned if the function is successful. Otherwise, the return value is EOF, which is defined in stdio.h.

For disk files, an EOF return usually means that the disk is full. However, this type of return can also occur if the device is write-protected or if a write error occurs. In any case, additional error information can be found in errno and _OSERR.

SEE

errno, fdopen, fopen, freopen, _OSERR, putc, putchar

fputs

Put a string to a file

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

error = fputs(s,fp);

int error;      non-zero if error
const char *s;  string pointer
FILE *fp;       file pointer
```

DESCRIPTION

The fputs function copies string s to a file that was previously opened for output via fopen, freopen, or fdopen. The string must be terminated by a null byte, which is not copied.

See puts for an example involving the fputs function.

RETURNS

If an error occurs, the return value is -1; otherwise, it is 0. Additional error information can be found in errno and _OSERR.

SEE

errno, ferror, fopen, fputc, puts

fputw, fputl

Put a word/longword to a buffered file

Class: UNIX

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

err = fputw(fp,x);
lerr = fputl(fp,y);

short err;      error value
long lerr;      error value
short x;        word to write to stream
long y;         longword to write to stream
FILE *fp;       file pointer
```

DESCRIPTION

The fputw and fputl functions write words and longwords respectively to the associated file. If the value cannot be written (typically because the disk is full), the value EOF cast to the appropriate type is returned. Note that it may not be possible to distinguish EOF from legitimate characters and so the value of feof and ferror should be checked in these cases.

Note that these functions produce files that are highly non-portable as they give no indication of the ordering of bytes on the machines architecture.

RETURNS

The functions return the value written to the stream or the value EOF if an I/O error occurs.

SEE

errno, feof, ferror, fgetc, fread, fgetw, fgetl

fread

Read blocks from a buffered file

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

a = fread(b,bsize,n,fp);

size_t a;       actual number of blocks
void *b;        pointer to first block
size_t bsize;   size of block in bytes
size_t n;       maximum number of blocks
FILE *fp;       file pointer
```

DESCRIPTION

The fread function performs buffered I/O operations to read blocks of data. Each block contains bsize bytes and up to n blocks are stored into contiguous memory locations beginning at location b.

For fread, blocks are read until n have been stored or until the end-of-file is hit. If the end-of-file is hit in the middle of a block, that partial block will be stored in the b array, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were read.

Note that in this implementation fread is implemented to be as fast as possible, hence for many applications the speed of fread will be better than the lower level read.

RETURNS

The fread function returns the number of complete blocks that were processed. A return value of -1 indicates that an error occurred, and further information about the error can be found in errno and _OSERR.

SEE

fclose, feof, ferror, fgetc, fopen, fputc, fseek, fwrite

free

Free a memory block

Class: ANSI

Category: Memory Management

SYNOPSIS

```
#include <stdlib.h>

free(b);

void *b; block pointer
```

DESCRIPTION

The free function releases a block that was previously obtained via calloc, malloc, or realloc.

SEE

calloc, malloc, realloc, getmem, rlsmem, sbrk

EXAMPLE

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

struct LIST
{
  struct LIST *next;
  char text[2];
};

int main(int argc,char *argv[])
{
  struct LIST *p;
  struct LIST *q;
  struct LIST list;
  char b[256];
  int x;

  for (;;)
  {
    printf("\nBegin new group...\n");
    for (q = &list; ; q = p)
    {
      printf("Enter a text string: ");
      if (!gets(b))
        break;
      if (b[0] == NULL)
      {
        if (q == &list)
          exit(0);
        break;
      }
```

```
        x = sizeof(struct LIST) - 2 +strlen(b) + 1;
        p = malloc(x);
        if (p == NULL)
        {
          printf("No more memory\n");
          break;
        }
        q->next = p;
        p->next = NULL;
        strcpy(p->text, b);
      }
      printf("\n\nTEXT LIST...\n");
      for (p = list.next; p != NULL; p = p->next)
      {
        printf("%s\n", p->text);
        free(p);
      }
      list.next = NULL;
    }

  return 0;
}
```


freopen

Reopen a buffered file

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

fpr = freopen(name, mode, fp);

FILE *fpr;          file pointer after re-opening
const char *name;   file name
const char *mode;   access mode
FILE *fp;           current file pointer
```

DESCRIPTION

This function reopens a buffered file. That is, it attaches a new file to a previously used file pointer. The previous file is automatically closed before the file pointer is reused. The name and mode arguments are the same as those for fopen.

RETURNS

The return file pointer, fpr, is NULL if an error occurred. Upon success, it is not guaranteed to be the same as fp. Specifically, it is an error to continue using fp after submitting that pointer to freopen.

SEE

fopen, fdopen

frexp — Split fraction and exponent

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <math.h>

f = frexp(v,xp);

double f;       fraction
double v;       value
int *xp;        exponent pointer
```

DESCRIPTION

The frexp function splits the floating point value v into its fraction (mantissa) and exponent parts. The mantissa is returned as a double whose absolute value is greater than or equal to 0.5 and less than 1.0. The exponent is returned as an integer whose absolute value is less than 1024.

SEE

fmod, ldexp, matherr, modf

fscanf — Formatted input from a file

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

n = fscanf(fp,fmt,arg1,arg2,...);

int n;              number of input items matched, or
                    EOF
FILE *fp;           file pointer
const char *fmt;    format string
void *argx;         pointers to input data areas
                    (x=1,2...)
```

DESCRIPTION

The fscanf function performs formatted input conversions on text obtained from a buffered file. The input characters are read and checked against the format string. The description of the scanf function fully describes the formats and conversion specifications.

RETURNS

The function returns the number of assignments that were made. For example, a return value of 3 indicates that conversion results were assigned to arg1, arg2, and arg3. If an end-of-file is reached before any values are assigned, the return value is EOF

SEE

cscanf, scanf, sscanf

fseek

Set buffered file position

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

error = fseek(fp,rpos,mode);

int error;      non-zero if error
FILE *fp;       file pointer
long int rpos;  relative file position
int mode;       seek mode
```

DESCRIPTION

The fseek function moves the byte cursor of a buffered file to a new position. The mode argument must be one of the following:

Mode	Meaning
SEEK_SET	The rpos argument is the number of bytes from the beginning of the file. This value must be positive.
SEEK_CUR	The rpos argument is the number of bytes relative to the current position. This value can be positive or negative.
SEEK_END	The rpos argument is the number of bytes relative to the end of the file. This value must be negative or zero.

The rewind macro resets the specified file to its first byte by means of a call to fseek.

RETURNS

A value of -1 is returned if an error occurs, with additional error information in errno and _OSERR.

A common programming error is to expect the return value to be equal to the current file position as with lseek.

SEE

errno, fgetpos, fopen, fsetpos, ftell, lseek, _OSERR, rewind, tell

fsetpos

Set file position indicator for stream

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

ret = fgetpos (strm,pos);

int ret;            0 if successful
FILE *strm;         stream
const fpos_t *pos;  file position info
```

DESCRIPTION

The fsetpos function sets the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which is the value obtained from an earlier call to the fgetpos function on the same stream.

A successful call to the fsetpos function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fsetpos call, the next operation on an update stream may be either input or output.

The fgetpos and fsetpos functions allow random access operations on files which are too large to handle with fseek and ftell.

RETURNS

If successful, the fsetpos function returns 0; on failure, the fsetpos function returns non-zero and stores an implementation-defined positive value in errno.

SEE

fgetpos

ftell

Get buffered file position

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

apos = ftell(fp);

FILE *fp;       file pointer
long int apos; absolute file position
```

DESCRIPTION

The ftell function returns a long value that is the current byte position in the file, relative to the beginning. In untranslated mode, it is equivalent to the following lseek call:

```
apos = lseek(fp->_file,0L,1);
```

In translated mode, ftell accounts for any removed carriage returns, giving a true offset into the physical file.

RETURNS

The ftell function returns a file position that can be used in a subsequent fseek call. An error is indicated by a return value of -1L. In this case, errno and _OSERR contain additional error information.

SEE

errno, fgetpos, fopen, fseek, fsetpos, lseek, _OSERR, rewind, tell

ftpack

Pack file time

Class: Lattice

Category: Date and Time

SYNOPSIS

```
#include <dos.h>

ft = ftpack(x);

long ft;            packed file time
const char *x;      unpacked file time
```

DESCRIPTION

The ftpack function packs the 32-bit value that GEMDOS uses in file descriptor blocks. The packed file time format is:

Bits	Contents
00-04	Second/2 (0 to 29)
05-10	Minute (0 to 59)
11-15	Hour (0 to 23)
16-20	Day (0 to 31)
21-24	Month (1 to 12)
25-31	Year-1980 (0 to 127)

The unpacked file time occupies a 6-byte array as follows:

Byte	Contents
0	Year - 1980
1	Month (1 to 12)
2	Day (1 to 31)
3	Hour (0 to 23)
4	Minute (0 to 59)
5	Second (0 to 59)

The getft and chgft functions can be used to get and change the packed time value for a particular file. Also, stpdate and stptime can be used to convert the unpacked file time into various ASCII forms. For example,

```
char  b[20],  x[6],  *p;

p  =  stpdate(b,2,x);
*p++  =  ' ';
p  =  stptime(p,2,&x[3]);
```

will convert the unpacked time value from x into an ASCII string such as 07/04/85 11:23:52.

RETURNS

The ftpack function returns the file time according to the packed file format given previously. No errors are returned, regardless of whether an invalid file time is supplied.

SEE

chgft, ftunpk, getft, stpdate, stptime

ftunpk

Unpack file time

Class: Lattice

Category: Date and Time

SYNOPSIS

```
#include  <dos.h>

ftunpk(ft,x);

long  ft;      packed  file  time
char  *x;      unpacked  file  time
```

DESCRIPTION

The ftunpk function unpacks the 32-bit value that GEMDOS uses to represent the time stamp on a file. See the description of ftpack for a complete description of the file time formats, packed and unpacked.

SEE

chgft, ftpack, getft, stpdate, stptime

fwrite

Write blocks to a buffered file

Class: ANSI

Category: Stream I/O

SYNOPSIS

```
#include <stdio.h>

a = fwrite(b,bsize,n,fp);

size_t a;         actual number of blocks
const void *b;    pointer to first block
size_t bsize;     size of block in bytes
size_t n;         maximum number of blocks
FILE *fp;         file pointer
```

DESCRIPTION

The fwrite function performs buffered I/O operations to write blocks of data. Each block contains bsize bytes and up to n blocks are written from contiguous memory locations beginning at location b.

For fwrite, blocks are written until n have been sent or until the output device cannot accept any more. If the output device becomes full in the middle of a block, a partial block will be written, but it will not be included in the function return value. In other words, the return value indicates the number of complete blocks that were written.

Note that in this implementation fwrite is implemented to be as fast as possible, hence for many applications the speed of fwrite will be better than the lower level write.

RETURNS

The fwrite function returns the number of complete blocks that were processed. A return value of 0 indicates a "no space" condition for fwrite. A return value of -1 indicates that an error occurred, and further information about the error can be found in errno and _OSERR.

SEE

fclose, feof, ferror, fgetc, fopen, fputc, fread, fseek

gcvt

Convert float to string

Class: UNIX

Category: Data Conversion/Formatting

SYNOPSIS

```
#include <math.h>

p = gcvt(v,dig,buffer);

char *p;          points to buffer
double v;         floating point value
int dig;          number of significant digits
char *buffer;     output buffer
```

DESCRIPTION

The gcvt function converts the specified floating point value into a null-terminated string in the output buffer. The string will be in either of two formats. First, gcvt attempts to produce dig significant digits in the FORTRAN F format. If that fails, it produces dig significant digits in the FORTRAN E format. Trailing zeroes will be eliminated if necessary.

Capabilities previously offered through ecvt, fcvt, and gcvt are now available by means of the ANSI function sprintf.

RETURNS

The function returns a pointer to the start of buffer, which you should ensure is large enough.

SEE

ecvt, fcvt

EXAMPLE

```
/*
 * This example displays 314150
 */
#include <math.h>
#include <stdio.h>

int main(void);
{
  char s[100];

  return printf("%s\n",gcvt(-3.1415e5,7,s));
}
```

geta4 — Establish addressability to the global data area

Class: Lattice *Category: Builtin Functions*

SYNOPSIS

```
#include <dos.h>

geta4();
```

DESCRIPTION

The geta4 function sets up the global data base register so that merged global data may be accessed. It is identical in function to compiling the subroutine with the -y option or putting the __saveds keyword on the declaration. It is provided only so that you do not need to change your code when using other compilers where you may provide a dummy geta4 routine. The -y option and __saveds keyword are preferred over geta4.

getc, getchar — Get a character

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

c = getc(fp);    get a character from a file
c = getchar();   get a character from stdin

int c;           return character or code
FILE *fp;        file pointer
```

DESCRIPTION

These functions get a single character from a file that was previously opened via fopen or fdopen. For getchar, the standard input file is read via file pointer (stdin). Note that getc and getchar are actually implemented as macros in order to maximise execution speed.

RETURNS

Upon success, the next input character is returned. Otherwise, the functions return EOF, which is defined in stdio.h.

In the event of an EOF return, error information can be found in errno and _OSERR. Most programmers treat any EOF return as an indication of end-of-file. However, if you want to distinguish an error from an end-of-file, you should reset errno before calling the function and then analyse its contents when you receive an EOF return.

SEE

fopen, errno, fgetc, fgetchar, fgets, gets, _OSERR

getcd

Get current directory

Class: GEMDOS

Category: DOS Interface

SYNOPSIS

```
#include <dos.h>

error = getcd(drive,path);

int error;   0 if successful
int drive;   drive code
char *path;  points to path area
```

DESCRIPTION

This function gets the current directory path for the specified disk drive. The drive codes are 0 for the current drive, 1 for drive A, 2 for drive B, and so on.

Note that the path area must be large enough to contain the expected path (FMSIZE is a safe value). The returned string will contain the entire path, including the drive name of the device.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

Dgetpath, getcwd, errno, _OSERR

getch, getche

Get char from console

Class: Lattice

Category: Console and Port I/O

SYNOPSIS

```
#include <dos.h>

c = getch();     get char from console (no echo)
c = getche();    get char from console (echo)

int c;           character obtained
```

DESCRIPTION

The getch and getche functions perform I/O operations with the keyboard and display attached as the console device. The getch function waits until a keyboard character is available and then returns it. The character is not displayed on the screen. To automatically echo each input character, use getche.

For the Atari ST and equivalent computers (e.g. IBM-PC), a return value of zero indicates that the keyboard character has no ASCII equivalent. The next call to getch or getche will then return the keyboard scan code.

Note that if you push back a non-ASCII scan code, the next call to getch or getche won't produce the usual zero return that indicates a scan code is coming.

RETURNS

As noted above.

SEE

cgets, cputs, kbhit, putch, ungetch

getclk

Get system clock

Class: Lattice

Category: DOS Interface

SYNOPSIS

```
#include <dos.h>

getclk(clock);

unsigned char *clock;
```

DESCRIPTION

The getclk function obtains the current setting of the system clock and places it into an 8-byte array as follows:

Byte	Contents
0	Day of week (0 for Sunday)
1	Year - 1980
2	Month (1 to 12)
3	Day (1 to 31)
4	Hour (0 to 23)
5	Minute (0 to 59)
6	Second (0 to 59)
7	Hundredths (0 to 99)

SEE

Tgetdate, Tgettime, chgclk, errno, _OSERR

getcwd

Get current working directory

Class: UNIX

Category: Process Environment

SYNOPSIS

```
#include <stdio.h>

p = getcwd(b,size);

char *p;        points to path buffer if successful,
                else NULL
char *b;        points to path buffer
size_t size;    size of path buffer
```

DESCRIPTION

This function obtains the path name for the current working directory. If the buffer pointer b is not NULL, then the path string is placed there if it will fit, and the return pointer p is the same as b. If b is NULL, then malloc is used to obtain a buffer of size bytes to hold the path string. In this latter case, you should use the free function to release the buffer when you are finished with it.

RETURNS

If the operation is successful, the function returns a pointer to the buffer. Otherwise it returns a NULL pointer and places error information in errno and _OSERR. Also, a NULL pointer is returned if the path string will not fit in the buffer or if a buffer cannot be allocated. In either of those cases, errno is unchanged, and _OSERR is reset.

SEE

getcd, errno, _OSERR

getdfs

Get free disk space

Class: GEMDOS

Category: Disk Functions

SYNOPSIS

```
#include <dos.h>

error = getdfs(drive,info);

int error;                  0 if successful
int drive;                  drive code
                            (0 => current drive)
struct DISKINFO *info;      disk information
```

DESCRIPTION

This function obtains information about the specified disk drive, including the amount of free space available. If a 0 is passed as the drive number, information is obtained about the current drive. The DISKINFO structure is defined in dos.h as follows:

```
struct DISKINFO
{
  unsigned long free; /* number of free clusters */
  unsigned long cpd;  /* clusters per drive */
  unsigned long bps;  /* bytes per sector */
  unsigned long spc;  /* sectors per cluster */
};
```

RETURNS

A return value of 0 indicates success. If the drive code is invalid or no disk is mounted on that drive, then the return value is -1. Additional information is provided in errno or _OSERR.

EXAMPLE

```
/*
 * Compute number of bytes available on current
 * drive
 */

#include <dos.h>

struct DISKINFO info;

long size;

if(getdfs(0,&info) == 0)
  size = (long)info.free * info.spc * info.bps;
```


getenv

Get environment variable

Class: ANSI

Category: Process Environment

SYNOPSIS

```
#include <stdlib.h>

var = getenv(name);

char *var;          environment variable pointer or
                    NULL
const char *name;   environment variable name
```

DESCRIPTION

This function searches the environment strings for one that has the form:

```
name=var
```

where name is the function argument. If such a string exists, the function returns a pointer to the var portion, which is null-terminated. Otherwise, a NULL pointer is returned.

RETURNS

As described above.

SEE

environ, putenv

EXAMPLE

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

char *path;

path = getenv("PATH");
if(path == NULL)
  fprintf(stderr,"No PATH variable\n");
else
  printf("%s\n",path);
```

getfa

Get file attribute

Class: GEMDOS

Category: File System Manipulation

SYNOPSIS

```
#include <dos.h>

fa = getfa(name);

int fa;                 file attribute or -1
const char *name;       file name
```

DESCRIPTION

This function gets the attribute byte for the specified file. The status is returned in fa and contains the following information:

Bit	Meaning
0	Read-only flag
1	Hidden file flag
2	System file flag
3	Volume label flag
4	Subdirectory flag
5	Archive flag (set if file has changed)
6	Reserved
7	Reserved

Note that the archive bit is only supported correctly in version 1.4 and above of the operating system.

RETURNS

If the operation is unsuccessful, the function returns -1 and places error information in errno and _OSERR.

SEE

Fattrib, errno, _OSERR

getfnl

Get file name list

Class: Lattice

Category: File Name Manipulation

SYNOPSIS

```
#include <stdlib.h>

n = getfnl(fnp,fna,fnasize,attr);

long n;                 number of matched files
const char *fnp;        file name pattern
char *fna;              file name array
size_t fnasize;         size of file name array
int attr;               file attribute
```

DESCRIPTION

This function gets all file names that match the specified pattern and attribute, and it places them into the file name array. Each name is stored as a null-terminated string, and the file name array is terminated by a null string (i.e., a string consisting of only a null byte). If the file name pattern includes a path prefix, that prefix is placed in front of each matching file name.

The function return value is the number of strings stored in the array, not including the terminating null string.

The file name pattern has the general form:

```
drive:path\node.ext
```

The function first strips off the drive and directory path portion and restricts its search to that area of the file system. The node and ext parts can contain any valid file name characters, including the * and ? pattern matching characters. Some examples are:

`"a:*.c"`	Finds all files on drive A that have ".c" as their extension. A file named "abc.c" would thus be place in the array as "a:abc.c".
`"\\abc\\def\\q*.x?"`	Finds all files in the directory \abc\def that begin with the letter q and have extensions consisting of the letter x and one other letter. For example, one such name would be "\\abc\\def\\queen.x". Note that the directory separator is actually a single backslash (\), but you must code it as a double backslash within the C string.

`"XYZ*."`	Finds all files in the current directory that begin with "XYZ" and have no extension. One example is "XYZ"

Notice that GEMDOS makes no distinction between upper and lower case in any part of the file name.

The attribute is a set of flag bits as follows:

Bit	Meaning (when set)
0	Read-only flag
1	Hidden file flag
2	System file flag
3	Volume label flag
4	Subdirectory flag
5	Archive flag (set if file has changed)
6	Reserved (must be zero)
7	Reserved (must be zero)

If all bits are reset, getfnl will find only normal files. If you want to include any of the other types, the appropriate flag must be set. For example, set bits 1 and 2 to find all matching normal, hidden, and system files. One special case is when bit 3 is set to specify a search for the volume label. That search will not find any file other than the label, regardless of how the other bits are set.

RETURNS

A value of -1 is returned if the file name pattern is invalid or if there is not enough room in the file name array. In the first case, _OSERR will contain further error information.

SEE

dfind, dnext, strbpl, strsrt, _OSERR

EXAMPLE

```
/*
 * This program constructs an array of pointers to
 * all normal files in the current directory that
 * have an extension of ".c". Then the array is
 * sorted into ASCII order.
 */

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

char  names[3000],*pointers[300];
int  count;

count  =  getfnl("*.c",names,sizeof(names),0);

if(count > 0)
{
  if(strbpl(pointers,300,names)  !=  count)
  {
    fprintf(stderr,"Too  many  file  names\n");
    exit(1);
  }
  strsrt(pointers,count);
}
else
{
  if(_OSERR)
    poserr("FILES");
  else
    fprintf(stderr,"Too  many  files\n");
  exit(1);
}
```

getft — Get file time

Class: GEMDOS *Category: File System Manipulation*

SYNOPSIS

```
#include <dos.h>

ft = getft(fh);

long ft;      file time or -1 if error;
int fh;       file handle
```

DESCRIPTION

This function gets the time and date information associated with the specified file. This information usually indicates when the file was created or last updated. It has the following format:

Bits	Contents
00-04	Second/2 (0 to 29)
05-10	Minute (0 to 59)
11-15	Hour (0 to 23)
16-20	Day (0 to 31)
21-24	Month (1 to 12)
25-31	Year-1980 (0 to 127)

RETURNS

If getft is successful, the file time (a long integer) is returned. Otherwise a value of -1L is returned. Additional error information can be found in errno and _OSERR.

SEE

Fdatime, chgft, errno, _OSERR

getmem, getml — Get a memory block

Class: OLD *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

p = getmem(sbytes);      Get small memory block
p = getml(lbytes);       Get large memory block

void *p;                 block pointer
unsigned sbytes;         number of bytes
size_t lbytes;           number of bytes
```

DESCRIPTION

These functions allocate a block and return a pointer to the first byte in the block. If the pool does not currently contain a block of sufficient size, the memory allocator obtains more space from the operating system. If that step fails, a NULL pointer is returned.

You will probably want to use the malloc function instead of getmem.

RETURNS

A NULL pointer is returned if the block could not be allocated. Otherwise, a character pointer is returned, but it can be cast to any other pointer type.

SEE

rlsmem, rlsml, sizmem

getopt

Get option letter from argument vector

Class: UNIX

Category: Argument Processing

SYNOPSIS

```
#include <stdlib.h>

c = getopt(argc,argv,optstring);

int c;                      argument character
int argc;                   argument count
const char *argv[];         argument vector
const char *optstring;      string containing valid opts

extern char *optarg;        pointer to option argument
extern int optind;          index of next argument
extern int opterr;          error message setting
```

DESCRIPTION

The getopt function returns the next option letter in argv which matches a letter in optstring. optstring contains all the option letters which are to be recognised, optionally followed by a colon (:) when an argument is required by the option. Such an argument may either be concatenated with the option letter, or be the next argument. The external variable optarg is set to point to any such argument.

The external variable optind is used to track the next argv index which getopt will use and is normally initialised to 1 by the first call to getopt.

When all options have been processed (i.e. the first argument which does not start with a '-'), or the special delimiter '--' has been encountered the value -1 is returned and and the '--' argument skipped.

When an unrecognised option is encountered, or an argument option is omitted where one was expected, an error message is printed on stderr and the value '?' returned. The printing of error messages may be disabled by setting the external variable opterr to 0.

Note that unlike argopt, getopt does *not* recognise a '/' as an option prefix.

RETURNS

The value of the character obtained as an option, '?' for an invalid option or -1 if no more arguments are available.

SEE

argopt, main

EXAMPLE

```
/*
 * parse the command lines:
 *   myprog -x -ypdq -z -g moo blah
 */

#include <stdlib.h>

int main(int argc, char *argv[])
{
  int c;
  char *file,*status;
  int x=0,z=0;

  while ((c=getopt(argc,argv,"xy:zg:"))!=-1)
    switch (c)
    {
      case 'x':
        x++;
        break;

      case 'z':
        z++;
        break;

      case 'y':
        status=optarg;
        break;

      case 'g':
        file=optarg;
        break;

      case '?':
        abort();
        break;
    }

  for (; optind<argc; optind++)
    process(argv[optind],x,z,status,file);

  return 0;
}
```

getpf, getpfe — Get program file

Class: Lattice

Category: Process Creation

SYNOPSIS

```
#include <dos.h>

error = getpf(file,prog);  Get program file
error = getpfe(file,prog); Get program file via
                           environment

int error;                 non-zero if error
char *file;                file name
const char *prog;          program name
```

DESCRIPTION

These functions find the loadable file that corresponds to the specified program name. The getpf function proceeds by first searching for the file "prog.PRG" then "prog.TTP", "prog.TOS" and "prog.APP". In each case, the access function is used to test for the file's existence. The getpfe functions uses the environment variable 'PATH' to search for the program file, in conjunction with the getpf function.

RETURNS

A non-zero value is returned if the file cannot be found.

The file argument must refer to an area that can hold the largest possible file name. The value FMSIZE is defined in dos.h for this purpose.

SEE

open, opene

EXAMPLE

```
/*
 * Find the file for program "myprog"
 *
 */

#include <stdio.h>
#include <dos.h>

char x[FMSIZE];

if(getpf(x,"myprog"))
  printf("Can't find program\n");
```


getpid — Get process identifier

Class: UNIX

Category: Process Environment

SYNOPSIS

```
#include <stdlib.h>

pid = getpid();

int pid;  process identifier
```

DESCRIPTION

This function returns a number that uniquely identifies the current process.

RETURNS

A integer uniquely identifying the process. Note that under GEMDOS this value has little significance unlike under multitasking systems.