Lattice C 5

the C Compiler for your Atari ST Computer

Volume II

Library Manual

Requires:

- ✓ Atari 520ST upwards
  (1M+ memory advised)
- ✓ Disk drive
  (2 floppies or hard disk advised)
- ✓ Mouse

HiSoft
High Quality Software

Lattice C

The C system for your Atari ST

Volume II
Library Manual

Published by HiSoft

Version 5
First edition March 1990 (ISBN 0 948517 30 1)
Second edition April 1991

ISBN for this volume 0 948517 38 7

ISBN for complete 3 volume set 0 948517 28 X

Set using an Apple Macintosh™ and Laserwriter™ with Microsoft Word™ and SuperPaint™.

Table of Contents

1 Introduction

This volume describes the Lattice C library, consisting of the Lattice portable library, the ANSI C library and the UNIX functions available to user programs. Note that this does not include the GEMDOS, BIOS, XBIOS, AES, VDI or Line-A functions which are documented separately in the volume 3.

The next section of this manual covers the header files supplied for use with the functions described in this manual. Many of the headers files are as defined by ANSI, but often contain extensions to provide a more flexible interface. Some of the header files are additions to ANSI and provide access to facilities available on the Atari ST in a more consistent manner than by directly calling the OS. The use of these functions greatly enhances portability to other Lattice C compilers.

The main section provides detailed descriptions of the library functions often with examples. All functions are described in the same basic way, with a synopsis, a description of the function as implemented, the input and output parameters and any side effects of the call, and finally any cross-references to other functions which are related or perform similar functions.

The synopses give a brief summary, listing the header file in which the function is declared, the calling syntax and the types of the parameters.

The calling form is listed as a one line summary, for instance fopen is:

```
#include <stdio.h>

fp = fopen(name, mode);
```

so that the function takes two parameters name and mode returning a single parameter. If the function does not return a value (i.e. 'returns void') then this is indicated by the return value not being assigned.

The types of parameters is then listed; note that the types listed are those used in the *definition*, to call them only compatible types are required. Hence considering fopen, the parameters are:

```
FILE *fp;                /* function return value in
                         appropriate type */
const char name;         /* first parameter */
const char *mode;        /* second parameter */
```

In general then the types of the parameters you pass would have type (char *) rather than (const char *).

Considering a more complex function such as qsort, the synopsis for this is:

```
#include <stdlib.h>

qsort(a,n,size,cmp);  Sort a data array

void *a;              data array pointer
size_t n;             number of elements in array
size_t size;          element size in bytes
int (*cmp)(const void *,const void *);
                      pointer to comparison function
```

So that qsort takes four parameters and returns no value. Examining the types of the parameters, the first has type (void *) whose type is compatible with any pointer type (i.e. you may pass any pointer). The second two parameters are of type size_t, hence in general these would passed values of type int. The size_t type was introduced by ANSI and is the type returned by the sizeof operator, (unsigned long) in this implementation. The final parameter is a functional parameter, which takes two pointers to constant objects.

The final form which appears in the synopses are those functions using the ANSI ellipsis operator to indicate a function which takes a variable number of arguments. For instance the printf function synopsis is:

```
#include <stdio.h>

length = printf(fmt,arg1,arg2,...);

const char *fmt;      format string
```

Hence printf takes a constant format string and a variable number of arguments relating to the formatting string. Note that when using variable argument functions you *must* ensure that you pass an appropriate type as the compiler is unable to check the types of your parameters and promote (or demote) them if necessary.

The fonts used throughout this library manual are:

OCRB	Program fragments and synopses.
Avante Garde	Library identifiers, parameters, disk files and keyboard shortcuts. Note that square brackets (i.e. those used in array accesses) appear as [ ] in this font, whereas parentheses (i.e. those used in function calls) appear as ( ). Beware of the distinction.

Note that *italics* are used solely for emphasis.

2 Header Files

This section describes the header files supplied with the Lattice C compiler, listing the header file and any macros, functions and types declared within them. To gain access to the facilities in these files you must #include them into your program.

For functions declared in a header file, the prototypes are listed so that you can see the types which the parameters should take and the value returned. In general description is not provided on these and you should refer to the main library section for full details.

Any types declared in a header file are listed together with their use. Note that many types were added by the ANSI standardisation committee and so may not be familiar, even to experienced C programmers.

Macros which provide a function like facility are listed in a functional form. Note that in general you may #undef these macros to obtain a true function implementation.

For macros which provide constant values these are indicated as 'const int' which will in general be the type assigned to these expressions. Note that it is important to be aware that these values are expressions and not simple variables as this can lead to unexpected type assignments (e.g. -32768 is the long integer value 32768 negated by the unary minus operator, giving a long integer type to the 'constant').

For external variables made available by a header these are marked as 'extern', and in general you may redefine this yourself to change the default initialiser, or alter them at runtime.

In the past many C programmers have neglected to include the required header files and simply placed a declaration in their own file. This practice is strongly discouraged, as ANSI changed the types of the parameters of many functions from the default int, hence your code may not run successfully without in-scope prototypes.

Class: ANSI *Category: Debugging*

SYNOPSIS

```
void assert(int);
```

DESCRIPTION

The assert.h header file contains the definition for the assert macro, which is used to insert diagnostics into a program during debugging, which can then be removed at final compilation time by defining the symbol NDEBUG, causing all references to assert to be removed during the pre-processing phase.

basepage.h

Program basepage definitions

Class: GEMDOS *Category: Process Environment*

SYNOPSIS

```
typedef struct _base
{
    void            *p_lowtpa;      bottom of TPA
    void            *p_hitpa;       top of TPA + 1
    void            *p_tbase;       base of text segment
    long            p_tlen;         length of text
    void            *p_dbase;       base of data segment
    long            p_dlen;         length of data
    void            *p_bbase;       base of BSS segment
    long            p_blen;         length of BSS
    void            *p_dta;         current DTA pointer
    struct _base    *p_parent;      parent's basepage
    void            *p_reserved;
    char            *p_env;         environment strings
    long            p_undef[20];
    char            p_cmdlin[128];  command line image
} BASEPAGE;

extern BASEPAGE *_pbase;  program's basepage pointer
```

DESCRIPTION

The basepage.h header file contains definitions relating to the GEMDOS basepage structure (usually called Program Segment Prefix (PSP) under MS-DOS).

The _pbase variable is used to gain access to the current program's basepage.

Note that this file is included by dos.h.

conio.h

Console I/O declarations

Class: GEMDOS *Category: Console and Port I/O*

SYNOPSIS

```
int  cget(void);
int  cgetc(void);
char  *cgets(char *);
int  cputc(int);
int  cputs(const char *);
int  cprintf(const char *, ...);
int  cscanf(const char *, ...);
int  getch(void);
int  getche(void);
int  kbhit(void);
int  iskbhit(void);
int  putch(int);
int  ungetch(int);
```

DESCRIPTION

The conio.h header file contains definitions for console input and output. These functions read and write characters directly at the GEMDOS level. They do not work through any layer of the file manager (i.e. buffered or unbuffered I/O) and so these functions will always return a key immediately one is requested, or write the character as soon as it is sent.

Note that traditionally these functions have been defined in the Lattice dos.h header file, and if you require portability to other Lattice compilers you should include dos.h rather than this file directly (which is included by dos.h anyway).

Class: ANSI *Category: Character Classification/Conversion*

SYNOPSIS

```
int  isalpha(int);      true if c is alpha
int  isupper(int);      true if c is upper case
int  islower(int);      true if c is lower case
int  isdigit(int);      true if c is a digit (0 to 9)
int  isxdigit(int);     true if c is a hexadecimal digit
                        (0 to 9, A to F, a to f)
int  isspace(int);      true if c is white space
int  ispunct(int);      true if c is punctuation
int  isalnum(int);      true if c is alpha or digit
int  isprint(int);      true if c is printable
                        (including blank)
int  isgraph(int);      true if c is graphic (excluding
                        blank)
int  iscntrl(int);      true if c is control character
int  isascii(int);      true if c is ASCII
int  iscsym(int);       true if valid character for C
                        symbols
int  iscsymf(int);      true if valid first character
                        for C symbols
int  _toupper(int);     convert lower case to upper case
int  _tolower(int);     convert upper case to lower case
int  toascii(int);      convert character to ascii
int  toupper(int);      convert character to upper case
int  tolower(int);      convert character to lower case
```

DESCRIPTION

The ctype.h header file contains macros for classifying (is...) and for converting characters (to...).

The is... functions return a non-zero value when the character falls into the category of the function. The to... functions return the character converted as required. Note that the is... functions are normally implemented as macros. If you wish to force the use of an equivalent function the macro should be undefined using #undef is....

Note that the functions isascii, iscsym, iscsymf, _toupper, _tolower, and toascii do not form part of the ANSI C standard.

Class: POSIX *Category: Directory Manipulation*

SYNOPSIS

```
struct dirent
{
    int d_attr;             /* GEMDOS file attribute */
    time_t d_time;          /* time */
    size_t d_size;          /* file size */
    char d_name[FMSIZE];    /* directory entry name */
};

typedef ... DIR;

DIR *opendir(const char *);
struct dirent *readdir(DIR *);
void closedir(DIR *);
void seekdir(DIR *,long );
long telldir(DIR *);
void rewinddir(DIR *);
```

DESCRIPTION

The dirent.h header file contains functions for manipulating directory entries in an OS independent manner. A directory is first opened using the openddir function and entries are then obtained from it using the readdir function. Seeking may also be performed in a manner similar to the buffered I/O subsystem using the seekdir, telldir and rewinddir functions.

The readdir command returns a pointer to the structure shown above, the only element of which you should rely upon being present is the d_name field. The GEMDOS specific entries are obviously not available under other operating systems.

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
const int SECSIZ;          disk sector size
const int FNSIZE;          maximum file node size
const int FMSIZE;          maximum file name size
const int FESIZE;          maximum file extension size

extern short _tos;         tos version number
extern short _country;     OS country code
extern long _MSTEP;        OS memory increment
extern long volatile _OSERR;
                           last OS error
extern unsigned long int _STACK;
                           default stack size

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 */
};

struct FILEINFO
{
   char resv[21];       /* reserved */
   char attr;           /* actual file attribute */
   long time;           /* file time and date */
   long size;           /* file size in bytes */
   char name[FNSIZE];   /* file name */
};

long _dclose(int);
long _dcreat(const char *, int);
long _dcreatx(const char *, int);
int _ddup(int);
int _ddup2(int, int);
int _disatty(int);
long _dopen(const char *, int);
long _dread(int,void *, long);
long _dwrite(int,const void *,long);
long _dseek(int, long, int);
int dfind(struct FILEINFO *, const char *, int);
int dnext(struct FILEINFO *);
int getcd(int, char *);
int getfa(const char *);
int chgfa(const char *, int);
int getdsk(void);
int chgdsk(int);
void chgdta(struct FILEINFO *);
struct FILEINFO *getdta(void);
int getdfs(int,struct DISKINFO *);
long getft(int);
int chgft(int, long);
long ftpack(const char *);
void ftunpk(long, char *);
```

```
int chgclk(unsigned char *);
void getclk(unsigned char *);
int getpf(char *,const char *);
int getpfe(char *,const char *);
__stdargs void _stub(void);
__stdargs void _xcovf(void);
int onbreak(int (*)());
int poserr(const char *);
void geta4(void);
void __emit(short);
long getreg(int);
void putreg(int, long);
const int REG_D0;          register D0 for getreg/putreg
const int REG_D1;          register D1 for getreg/putreg
const int REG_D2;          register D2 for getreg/putreg
const int REG_D3;          register D3 for getreg/putreg
const int REG_D4;          register D4 for getreg/putreg
const int REG_D5;          register D5 for getreg/putreg
const int REG_D6;          register D6 for getreg/putreg
const int REG_D7;          register D7 for getreg/putreg
const int REG_A0;          register A0 for getreg/putreg
const int REG_A1;          register A1 for getreg/putreg
const int REG_A2;          register A2 for getreg/putreg
const int REG_A3;          register A3 for getreg/putreg
const int REG_A4;          register A4 for getreg/putreg
const int REG_A5;          register A5 for getreg/putreg
const int REG_A6;          register A6 for getreg/putreg
const int REG_A7;          register A7 for getreg/putreg
```

DESCRIPTION

The dos.h header file contains functions for interfacing with GEMDOS, some of the internal library structures, and OS specific constants.

The major functions supplied by this library are the _d... functions which provide the libraries' interface to GEMDOS, resolving many of the anomalies encountered in manipulating GEMDOS directly.

Assorted file system manipulation functions are also provided together with the facilities to map GEMDOS return values into the standard library structures, via functions such as ftunpk, getclk etc.

This header file also includes the conio.h, basepage.h and osbind.h headers files for convenience.

Class: ANSI *Category: Errors*

SYNOPSIS

```
extern int volatile errno;    UNIX error number
extern int sys_nerr;          number of error codes
extern char *sys_errlist[];   UNIX error messages
const int E...;               error names
```

DESCRIPTION

The errno.h header file contains the ANSI errno variable which gives details of the last error encountered by the runtime library.

The sys_nerr and sys_errlist items give a count of the errors which may be produced, and a list of the error messages which the values of errno correspond to. Several macros (E...) are provided to give meaningful names to the error numbers produced and these are identical to those produced under UNIX.

Note that the sys_nerr, sys_errlist variable and the E... macros do *not* form part of the ANSI C standard.

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
int  open(const char *, int, ...);
int  opene(const char *, int, int, char *);
long  read(int, void *, size_t);
long  write(int, const void *, size_t);
int  creat(const char *, int);
long  lseek(int, long, int);
long  tell(int);
int  close(int);
int  iomode(int, int);
int  isatty(int);
long  filelength(int);

int  rename(const char *,const char *);
int  remove(const char *);
int  unlink(const char *);

const  int  O_RDONLY;   open in read only mode
const  int  O_WRONLY;   open in write only mode
const  int  O_RDWR;     open in read/write mode
const  int  O_APPEND;   allow only appends
const  int  O_CREAT;    creat file if absent
const  int  O_TRUNC;    truncate file if present
const  int  O_EXCL;     exclusive create flag
const  int  O_RAW;      open in untranslated mode

const  int  S_IREAD;    allow read access
const  int  S_IWRITE;   allow write access
const  int  S_IEXEC;    allow execute access
```

DESCRIPTION

The fcntl.h header file contains the interface definitions for the unbuffered I/O sub-system. The open, read, write, creat, lseek, tell, close, iomode, isatty and filelength functions manipulate a file given a library file handle. Note that the handles used by these functions are *not* GEMDOS file handles and you should use the chkufb function (defined in ios1.h) for access to the GEMDOS handle.

Several macros (O_... and S_...) are also defined in this file for use with the creat and open functions.

Note that this header file and its associated defintions do *not* form part of the ANSI C standard.

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
const int FLT_GUARD;
const int FLT_NORMALIZE;
const int FLT_RADIX;
const int FLT_ROUNDS;

const int DBL_DIG;
const double DBL_EPSILON;
const int DBL_MANT_DIG;
const double DBL_MAX;
const int DBL_MAX_10_EXP;
const int DBL_MAX_EXP;
const double DBL_MIN;
const int DBL_MIN_10_EXP;
const int DBL_MIN_EXP;

const int FLT_DIG;
const float FLT_EPSILON;
const int FLT_MANT_DIG;
const float FLT_MAX;
const int FLT_MAX_10_EXP;
const int FLT_MAX_EXP;
const float FLT_MIN;
const int FLT_MIN_10_EXP;
const int FLT_MIN_EXP;

const int LDBL_DIG;
const long double LDBL_EPSILON;
const int LDBL_MANT_DIG;
const long double LDBL_MAX;
const int LDBL_MAX_10_EXP;
const int LDBL_MAX_EXP;
const long double LDBL_MIN;
const int LDBL_MIN_10_EXP;
const int LDBL_MIN_EXP;

const double HUGE_VAL;
```

DESCRIPTION

The float.h header file contains macros giving the limits placed on the accuracy of floating point calculations. A floating point number is defined by:

$$x = s * b^{e} * \sum_{k=1}^{p} f_k * b^{-k},\ e_{\min} \le e \le e_{\max}$$

where s represents the sign, b the base of the exponent, e the exponent, p the precision of the mantissa (i.e. the number of digits in base b) and f_k the digits of the mantissa.

The prefixes FLT, DBL and LDBL refer respectively to float, double and long double, and the remaining part of the common expressions signify:

_DIG	The number of *decimal* digits of precision available int the appropriate type.
_EPSILON	The smallest number x such $1.0 + x$ is not equal to 1.0.
_MANT_DIG	Number of digits in the floating point mantissa in base FLT_RADIX (i.e. p in the above expression).
_MIN	The smallest absolute number expressible in the appropriate type.
_MIN_EXP	The smallest integer such that the value of FLT_RADIX raised to its power minus 1 is greater than or equal to _MIN.
_MIN_10_EXP	The smallest integer such that 10 raised to its power minus 1 is greater than or equal to _MIN.
_MAX	The largest number expressible in the appropriate type.
_MAX_EXP	The largest integer such that the value of FLT_RADIX raised to its power minus 1 is less than or equal to _MAX.
_MAX_10_EXP	The largest integer such that 10 raised to its power minus 1 is less than or equal to _MAX.

The remaining definitions are:

FLT_GUARD	Determines whether guard digits are used during multiplication. 0 indicates no, 1 indicates yes.
FLT_NORMALIZE	States whether normalisation is required for floating point quantities. 0 indicates no, 1 indicates yes.
FLT_RADIX	The radix of the exponent in the implementation (i.e. the value of b in the above expression).

FLT_ROUNDS	Type of rounding performed during conversion: -1 Indeterminate. 0 Toward zero (truncation). 1 To nearest. 2 To +∞ (i.e. always up). 3 To -∞ (i.e. always down).

The final definition in float.h is HUGE_VAL, this is normally defined in math.h but is duplicated here for convenience.

Note that the FLT_GUARD and FLT_NORMALIZE macros do *not* form part of the ANSI C standard, also note that ANSI places HUGE_VAL in math.h, not in float.h.

Class: Lattice *Category: Low-Level I/O*

SYNOPSIS

```
const int NUFBS;     default number of UNIX file
                     blocks

extern int _iomode;  default unbuffered mode
extern int _nufbs;   number of ubs allocated

struct UFB *chkufb(int);
```

DESCRIPTION

The ios1.h header file contains environment definitions for the unbuffered I/O sub-system. The value NUFBS contains the default number of handles which the libraries make available, this value is normally the same as _nufbs. The _iomode variable allows the default translation mode to be changed, whilst chkufb allows translation of library handles to GEMDOS handles.

Note that this header file and its associated defintions do *not* form part of the ANSI C standard.

Class: ANSI *Category: Process Environment*

SYNOPSIS

```
const  CHAR_BIT;     bits per char

const  CHAR_MAX;     max value for char
const  CHAR_MIN;     min value for char
const  SCHAR_MAX;    max value for signed char
const  SCHAR_MIN;    min value for signed char
const  UCHAR_MAX;    max value for unsigned char

const  SHRT_MAX;     max value for short int
const  SHRT_MIN;     min value for short int
const  USHRT_MAX;    max value for unsigned short int

const  INT_MAX;      max value for short int
const  INT_MIN;      min value for short int
const  UINT_MAX;     max value for unsigned short int

const  LONG_MAX;     max value for long int
const  LONG_MIN;     min value for long int
const  ULONG_MAX;    max value for unsigned long int

const  MB_LEN_MAX;   maximum bytes in a multibyte
                     character
```

DESCRIPTION

The limits.h header file contains macros defining the integral numerical limits of the program environment. Note that the values of CHAR_MAX and CHAR_MIN are dependent on the -cu flag, whilst INT_MAX and INT_MIN are dependent on the -w compiler flag.

You should be aware that these values are numeric constants and are assigned types according to the normal assignment rules, and hence do *not* neccesarily have the type of the limit they represent.

Class: ANSI *Category: Localisation*

SYNOPSIS

```
const int LC_COLLATE;    collation information
const int LC_CTYPE;      character handling
const int LC_MONETARY;   monetary information
const int LC_NUMERIC;    numeric information
const int LC_TIME;       time information
const int LC_ALL;        all information

struct lconv { ... };    monetary conversion
                         information

extern char DECPT;       local decimal point character

char *setlocale(int, const char *);
struct lconv *localeconv(void);

typedef ... wchar_t;     wide character type
```

DESCRIPTION

The locale.h header file declares functions and macros used for manipulating a program's locale. Note that the ANSI places the type wchar_t in the stddef.h and stdlib.h headers files, and its declaration here is for convenience.

Note that the lconv structure is documented under the localeconv function in the main part of this manual, whilst the LC... macros are discussed under setlocale.

Class: Lattice *Category: Mathematics*

SYNOPSIS

```
double  acos(double);    arc-cosine
double  asin(double);    arc-sine
double  atan(double);    arc-tangent
double  cos(double);     cosine
double  cosh(double);    hyperbolic cosine
double  exp(double);     exponential
double  fabs(double);    absolute value
double  fatanh(double);  hyperbolic arc-tangent
double  fetoxm1(double); exponential - 1
double  fgetexp(double); get exponent
double  fgetman(double); get mantissa
double  fintrz(double);  integer part, round to zero
double  flog2(double);   log base 2
double  flognp1(double); log (n+1)
double  fneg(double);    negate
double  ftentox(double); 10 to x
double  log(double);     log
double  log10(double);   log base 10
double  pow2(double);    2 to x
double  sin(double);     sine
double  sinh(double);    hyperbolic sine
double  sqrt(double);    square root
double  tan(double);     tangent
double  tanh(double);    hyperbolic tangent
```

DESCRIPTION

The m68881.h header file declares functions for the standard transcendental functions implemented by the M68881 which take a single argument. Note that the use of these functions requires use of -f8 on lc1, also the code generated *requires* a 68020, 68030 or suitable Line-F emulator to run., Specifically it will *not* work the Atari I/O mapped M68881 card.

Note that this header file and some its defintions do *not* form part of the ANSI C standard.

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
const double HUGE_VAL;

double acos(double);
double asin(double);
double atan(double);
double atan2(double, double);
double ceil(double);
double cos(double);
double cosh(double);
double exp(double);
double fabs(double);
double floor(double);
double fmod(double, double);
double frexp(double, int *);
double ldexp(double, int);
double log(double);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sin(double);
double sinh(double);
double sqrt(double);
double tan(double);

struct exception
{
   int     type;           error type
   char    *name;          math function name
   double  arg1, arg2;     function arguments
   double  retval;         proposed return value
};

const int DOMAIN;       domain error
const int OVERFLOW;     overflow
const int PLOSS;        partial loss of significance
const int RANGE;        range error
const int SING;         singularity
const int TLOSS;        total loss of significance
const int UNDERFLOW;    underflow

const int FPECOM;       not comparable
const int FPENAN;       not a number
const int FPEOVF;       overflow
const int FPEUND;       underflow
const int FPEZDV;       zero divisor

extern int _FPERR;      low-level floating point error
                        status

const double PI;        PI
const double PID2;      PI/2
const double PID4;      PI/4
const double I_PI;      1/PI
const double I_PID2;    1/(PI/2)
```

```
const double HUGE;     largest representable absolute
                       double
const double TINY;     smallest representable absolute
                       double
const double LOGHUGE; ln(HUGE);
const double LOGTINY; ln(TINY);

double cot(double);
double drand48(void);
double erand48(unsigned short *);
double except(int, char *, double, double, double);
char *ecvt(double, int, int *, int *);
char *fcvt(double, int, int *, int *);
char *gcvt(double, int, char *);
long jrand48(unsigned short *);
void lcong48(unsigned short *);
long lrand48(void);
int matherr(struct exception *);
long mrand48(void);
long nrand48(unsigned short *);
double pow2(double);
unsigned short *seed48(unsigned short *);
void srand48(long);
double tanh(double);
```

DESCRIPTION

The math.h header file declares functions and macros for the mathematical functions.

Note that some of the defintions in this header do *not* form part of the ANSI C standard.

oserr.h

TOS error definitions

Class: GEMDOS *Category: Errors*

SYNOPSIS

```
extern int volatile _OSERR;   TOS error number
extern int os_nerr;           number of error codes
extern char *os_errlist[];    TOS error messages
const int E...;               error names
```

DESCRIPTION

The oserr.h header file contains the operating system error variable _OSERR, which gives details of the last OS error encountered by the runtime library.

The os_nerr and os_errlist items give a count of the errors which may be produced, and a list of the error messages which the values of _OSERR correspond to. Several macros (E...) are provided to give meaningful names to the error numbers produced.

Note that this header file and its associated definitions do *not* form part of the ANSI C standard.

Class: ANSI *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
typedef ... jmp_buf;          jump buffer type

int  setjmp(jmp_buf);
void  longjmp(jmp_buf,int);
```

DESCRIPTION

The setjmp.h header file contains the declarations for non-local jumps. You should be aware of the potential problems using these functions, discussed under the main setjmp entry in this manual.

Class: ANSI *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
const int SIGABRT;   abnormal termination, abort()
const int SIGFPE;    floating point exception
const int SIGILL;    illegal instruction
const int SIGINT;    interrupt from GEMDOS
const int SIGSEGV;   segmentation violation
const int SIGTERM;   termination request

void (*SIG_DFL)(int); default action
void (*SIG_IGN)(int); ignore the signal
void (*SIG_ERR)(int); error return

void (*signal(int,void (*)(int)))(int);
int raise(int);

typedef ... sig_atomic_t; signal atomic type
```

DESCRIPTION

The signal.h header file contains the definitions and declarations for signal handling. Note that the signals provided are those required by ANSI however these are not necessarily called at any other time than explicitly via raise.

The type sig_atomic_t is a type which is guaranteed to be accessed atomically if simultaneous signals occur, however any variable definition *must* include the volatile modifier viz:

```
volatile sig_atomic_t sig_count;
```

Class: ANSI *Category: Variable Argument Handling*

SYNOPSIS

```
typedef ... va_list; variable list type

void  va_start(va_list,typename);
typename  *va_arg(va_list,typename);
void  va_end(va_list)
```

DESCRIPTION

The stdarg.h header file contains routines for manipulating variable numbers of arguments in an ANSI fashion. Note that the header file varargs.h provides similar facilities (and under similar names), but follows the UNIX definition.

EXAMPLE

```
/*
 * concatenate a variable number of strings,
 * terminated by NULL into a malloced block
 * of memory
 */

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

char *strcatl(const char *s1, ...)
{
  va_list strings;
  size_t length;
  char *s, *concat;

  va_start(strings, s1);
  length=strlen(s1);        /* fetch length of first
                               string */
  while (s=va_arg(strings, char *))
    length+=strlen(s);      /* add in remaining string
                               lengths */
  va_end(strings);          /* all done this pass */

  if (concat=malloc(length+1)) /* get some RAM */
  {
    va_start(strings, s1);
    strcpy(concat, s1);     /* copy first string */
    while (s=va_arg(strings, char *))
      strcat(concat, s);    /* concatenate rest */
    va_end(strings);
  }
  return concat; /* return composite string */
}
```

Class: ANSI *Category: Process Environment*

SYNOPSIS

```
typedef ... size_t;       type of sizeof
typedef ... ptrdiff_t;    type of pointer difference
typedef ... wchar_t;      wide character type

size_t offsetof(type,memb); obtain field offset

void *NULL;                 NULL pointer constant
```

DESCRIPTION

The stddef.h header file contains ANSI definitions for the types of compiler and library quantities.

The offsetof macro may be used for obtaining the byte offset of a field within an aggregate item.

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
typedef ... FILE;       FILE type
typedef ... fpos_t;     file position type

const int FILENAME_MAX;    max chars in a filename
const int FOPEN_MAX;       max number of open files

const int _IOFBF;       fully buffered flag
const int _IONBF;       non-buffered flag
const int _IOLBF;       line-buffered flag

const int BUFSIZ;       standard buffer size
const int EOF;          end-of-file code
const int L_tmpnam;     maximum tmpnam filename length

const int SEEK_SET;     seek from beginning of file
const int SEEK_CUR;     seek from current file position
const int SEEK_END;     seek from end of file

const int TMP_MAX;      maximum unique temporary files

FILE *stdin;            standard input file pointer
FILE *stdout;           standard output file pointer
FILE *stderr;           standard error file pointer
FILE *stdaux;           standard auxiliary file pointer
FILE *stdprt;           standard printer file pointer

int rename(const char *,const char *);
int remove(const char *);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *);
int fflush(FILE *);
FILE *fopen(const char *, const char *);
FILE *freopen(const char *, const char *, FILE *);
void setbuf(FILE *, char *);
int setvbuf(FILE *, char *, int, size_t);
int fprintf(FILE *, const char *, ...);
int fscanf(FILE *, const char *, ...);
int printf(const char *, ...);
int scanf(const char *, ...);
int sprintf(char *, const char *, ...);
int sscanf(const char *, const char *, ...);
int vfprintf(FILE *, const char *, va_list);
int vprintf(const char *, va_list);
int vsprintf(char *, const char *, va_list);

int fgetc(FILE *);
char *fgets(char *, int, FILE *);
int fputc(int, FILE *);
int fputs(const char *, FILE *);
int getc(FILE *);
int getchar(void);
char *gets(char *);
int putc(int, FILE *);
```

```
int putchar(int);
int puts(const char *);
int ungetc(int, FILE *);
size_t fread(void *, size_t, size_t, FILE *);
size_t fwrite(const void *, size_t, size_t, FILE *);
int fgetpos(FILE *, fpos_t *);
int fseek(FILE *, long int, int);
int fsetpos(FILE *, const fpos_t *);
long int ftell(FILE *);
void rewind(FILE *);
void clearerr(FILE *);
int feof(FILE *);
int ferror(FILE *);
void perror(const char *);

int fcloseall(void);
FILE *fdopen(int, const char *);
int fgetchar(void);
int fileno(FILE *);
int flushall(void);
void fmode(FILE *, int);
int fputchar(int);
int setnbf(FILE *);
int access(const char *, int);
int chdir(const char *);
int chmod(const char *, int);
char *getcwd(char *, int);
int mkdir(const char *);
int rmdir(const char *);
FILE *fopene(const char *, const char *,char *);
int unlink(const char *);

char *mktemp(char *s);
short fputw(short,FILE *);
long fputl(long,FILE *);
short fgetw(FILE *);
long fgetl(FILE *);

extern unsigned long __fmask; default file mask
extern int _fmode;    default access mode
extern int _bufsiz;   default file buffer size
```

DESCRIPTION

The stdio.h header file contains definitions, declarations and macros for use by the ANSI standard input/output library.

The following functions and variables which appear in this header do not form part of the ANSI C standard: __fmask, _bufsiz, _fmode, access, chdir, chmod, fcloseall, fdopen, fgetchar, fgetl, fgetw, fileno, flushall, fmode, fopene, fputchar, fputl, fputw, getcwd, mkdir, mktemp, rmdir, setnbf and unlink.

Class: ANSI *Category: General Functions*

SYNOPSIS

```
extern char MB_CUR_MAX;

typedef ... div_t;        div() type

typedef ... ldiv_t;       ldiv() type

void *malloc(size_t);
void *calloc(size_t,size_t);
void *realloc(void *, size_t);
void free(void *);

void *getml(size_t);
int rlsml(void *, size_t);
size_t sizmem(void);
size_t chkml(void);

void *getmem(unsigned);
int rlsmem(void *, unsigned);

void *alloca(size_t);

extern size_t _stkdelta;  stack/data chicken factor

void *sbrk(unsigned);
void *lsbrk(long);

int chdir(const char *);
int chmod(const char *, int);
char *getcwd(char *, int);
int mkdir(const char *);
int rmdir(const char *);

void qsort(void *, size_t, size_t,
            int (*)(const void *, const void *));
void dqsort(double *,size_t);
void fqsort(float *,size_t);
void lqsort(long *,size_t);
void sqsort(short *,size_t);
void tqsort(char **,size_t);

void bsearch(const void *, const void *, size_t,
        size_t, int (*)(const void *, const void *));

int mblen(const char *,size_t);
size_t mbstowcs(wchar_t *, const char *, size_t);
int mbtowc(wchar_t *, const char *, size_t);
size_t wcstombs(char *, const wchar_t *, size_t);
int wctomb(char *, wchar_t);
```

```
void exit(int);
void abort(void);
int atoi(const char *);
double atof(const char *);
long int atol(const char *);
char *getenv(const char *);

void _exit(int);
void _XCEXIT(int);
char *argopt(int, char *[], char *, int *, char *);

void *lsearch(const void *, void *, size_t *, size_t,
              int (*)(const void *, const void *));
void *lfind(const void *, const void *,
            const size_t *, size_t,
            int (*)(const void *, const void *));

int getpid(void);
int getopt(int argc, const char *argv[],
                      const char *optstring);

extern int optopt,opterr,optind;
extern char *optarg;

int system(const char *);
size_t _hash(const char *);

int abs(int);

long atol(char *);
char *ecvt(double, int, int *, int *);
char *fcvt(double, int, int *, int *);
char *gcvt(double, int, char *);
long getfnl(const char *, char *, size_t, int);
int iabs(int);
long labs(long);
int onexit(int(*)(int));
int putenv(char *);

int rand(void);
int rmvenv(const char *);
void srand(unsigned int);
double strtod(const char *,const char **);
long int strtol(const char *, char **, int);
unsigned long int strtoul(const char *,char **,int);
long int utpack(const char *);
void utunpk(long int, char *);

int atexit(void (* )(void));
div_t div(int, int);
ldiv_t ldiv(long int, long int);

unsigned long _lrotl(unsigned long,int);
unsigned short _rotl(unsigned short,int);
unsigned long _lrotr(unsigned long,int);
unsigned short _rotr(unsigned short,int);
```

```
int forkl(const char *,...);
int forkle(const char *,...);
int forklp(const char *,...);
int forklpe(const char *,...);
int forkv(const char *,const char **);
int forkve(const char *,const char **,const char **);
int forkvp(const char *,const char **);
int forkvpe(const char *,const char **,
                              const char **);

int wait(void);

const int EXIT_SUCCESS;   success exit value
const int EXIT_FAILURE;   failure exit value;

const int RAND_MAX;  maximum rand() value
```

DESCRIPTION

The stdlib.h header file contains general utility definitions, declarations and macros defined by the ANSI standard.

The following functions and variables which appear in this header do not form part of the ANSI C standard: _exit, _hash, _lrotl, _lrotr, _rotl, _rotr, _stkdelta, _XCEXIT, alloca, argopt, chdir, chkml, chmod, dqsort, ecvt, fcvt, forkl, forkle, forklp, forklpe, forkv, forkve, forkv, forkvpe, fqsort, gcvt, getcwd, getfnl, getmem, getml, getopt, getpid, labs, lfind, lqsort, lsbrk, lsearch, mkdir, onexit, optarg, opterr, optind, optopt, putenv, rlsmem, rlsml, rmdir, rmvenv, sbrk, sizmem, sqsort, tqsort, utpack, utunpk and wait.

Class: ANSI *Category: String manipulation*

SYNOPSIS

```
extern char _SLASH;  path separator character

char *strcat(char *, const char *);
char *strchr(const char *, int);
int strcmp(const char *, const char *);
char *strcpy(char *, const char *);
size_t strcspn(const char *, const char *);
size_t strspn(const char *, const char *);
size_t strlen(const char *);
char *strncat(char *, const char *, size_t);
int strncmp(const char *, const char *, size_t);
char *strncpy(char *, const char *, size_t);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
char *strstr(const char *, const char *);
char *strtok(char *, const char *);
char  *strerror(int);
int strcoll(const char *, const char *);
size_t strxfrm(char *, const char *, size_t);

size_t stcarg(const char *, const char *);
size_t stccpy(char *, const char *, size_t);
char *stpcpy(char *, const char *);
char *strdup(const char *);
void strins(char *, const char *);
char *strnset(char *, int, size_t);
char *strrev(char *);
size_t stcis(const char *, const char *);
size_t stcisn(const char *, const char *);
size_t stcpm(const char *, const char *, char **);
size_t stcpma(const char *, const char *);
char *stpblk(const char *);
char *stpbrk(const char *, const char *);
char *stpchr(const char *, int);
char *stpsym(const char *, char *, size_t);
char *stpchrn(const char *, int);
char *stptok(const char *, char *, size_t,
                                    const char *);
long strbpl(char **, size_t, const char *);

int stcd_i(const char *, int *);
int stcd_l(const char *, long *);
int stch_i(const char *, int *);
int stch_l(const char *, long *);
int stci_d(const char *, int);
int stci_h(const char *, int);
int stci_o(const char *, int);
int stcl_d(const char *, long);
int stcl_h(const char *, long);
int stcl_o(const char *, long);
int stco_i(const char *, int *);
int stco_l(const char *, long *);
```

```
int stcgfe(char *, char *);
int stcgfn(char *, char *);
int stcgfp(char *, const char *);

int stcsma(char *, char *);
int stcu_d(char *, unsigned);
int stcul_d(char *, unsigned long);
size_t stclen(const char *);
char *stpdate(char *, int, char *);
char *stptime(char *, int, char *);
int strmid(const char *, char *, size_t, size_t);
char *strlwr(char *);
void strmfe(char *, const char *, const char *);
void strmfn(char *, const char *, const char *,
                        const char *, const char *);
void strmfp(char *, const char *, const char *);
int strnicmp(const char *, const char *, size_t);
int stricmp(const char *, const char *);

char *strset(char *, int);
void strsfn(const char *, char *, char *, char *,
                                           char *);
char *strupr(char *);
int stspfp(char *, int *);
void strsrt(char *[], size_t);

void *memchr(const void *, int, size_t);
int memcmp(const void *, const void *, size_t);
void *memcpy(void *, const void *, size_t);
void *memmove(void *, const void *, size_t);
void *memset(void *, int, size_t);
void *memccpy(void *, const void *, int, size_t);
void *memswp(void *, void *, size_t);
void *memrep(void *, void *, size_t, size_t);
void setmem(void *, unsigned, int);
void movmem(void *, void *, unsigned);
void repmem(void *, void *, unsigned, unsigned);
void swmem(void *, void *, unsigned);
```

DESCRIPTION

The string.h header file contains the definitions for handling strings and buffers via the standard library.

This header file contains many functions which do not form part of the ANSI C standard, the functions which *do* appear therein are: memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok and strxfrm.

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
typedef ... time_t;   type returned by time()
typedef ... clock_t;  type returned by clock()

const int CLK_TCK;         clock() granularity
const int CLOCKS_PER_SEC;  clock() granularity

struct tm
{
  int tm_sec;     /* seconds after the minute */
  int tm_min;     /* minutes after the hour */
  int tm_hour;    /* hours since midnight */
  int tm_mday;    /* day of the month */
  int tm_mon;     /* months since January */
  int tm_year;    /* years since 1900 */
  int tm_wday;    /* days since Sunday */
  int tm_yday;    /* days since January 1 */
  int tm_isdst;   /* Daylight Savings Time flag */
};

clock_t clock(void);
double difftime(time_t, time_t);
time_t mktime(struct tm *);
time_t time(time_t *);
char *asctime(const struct tm *);
char *ctime(const time_t *);
struct tm *gmtime(const time_t *);
struct tm *localtime(const time_t *);
size_t strftime(char *, size_t, const char *,
                                  const struct tm *);

void getclk(unsigned char *);
int  chgclk(unsigned char *);
void utunpk(long, char *);
long utpack(const char *);

void _tzset(void);

extern int   __daylight;   daylight time flag
extern long  __timezone;   seconds from GMT
extern char  *__tzname[2]; time zone names
extern char  __tzstn[4];   standard time name
extern char  __tzdtn[4];   daylight time name

extern char *_TZ;     string for user time zone
```

DESCRIPTION

The time.h header file contains functions and macros for manipulating time in both internal and external representations.

Note that although ANSI defines this header file the getclk, chgclk, utunpk and utpack do not appear as part of the standard.

3 Library Functions

This section gives detailed descriptions of the library functions supplied with the Lattice C compiler, listing the header file in which the function is declared, the calling syntax and any parameters which should be supplied to the function.

As mentioned earlier, each entry consists of a synopsis, description and cross-reference. Also a 'Class' and 'Category' are listed giving the origin of the function, e.g. ANSI, Lattice, UNIX etc., and a category showing which family of functions the function falls into, e.g. Stream I/O, Date and Time.

In the past many C programmers have neglected to include the required header files and simply placed a declaration in their own file. This practice is strongly discouraged as ANSI changed the types of the parameters of many functions from the default `int`, hence your code may not run successfully without in-scope prototypes.

abort

Abort the current process

Class: ANSI *Category: Process Creation*

SYNOPSIS

```
#include <stdlib.h>

abort();
```

DESCRIPTION

This function aborts the current process and returns a completion code of 3 to the parent process. Also the message "Abnormal program termination" is sent to stderr. I/O buffers created via fopen are not flushed. Prior to termination the signal SIGABRT is asserted, as if by the call:

```
raise(SIGABRT);
```

RETURNS

The function does not return.

SEE

onexit, exit, _exit, raise

EXAMPLE

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

void validate(int x,int lower,int higher)
{
  if (x<lower || x>higher)
  {
    puts("Internal range check failed");
    abort();
  }
}
```

abs

Absolute value

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

ax = abs(x);

int x;        numeric data type
int ax;       absolute value of x
```

DESCRIPTION

The abs function computes the absolute value of the integer argument. Compare abs with the fabs function, which computes the absolute value of a float or a double, returning a double result.

Note that this function is normally implemented as the inline function __builtin_abs.

SEE

fabs, iabs, labs

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <stdio.h>

ret = access(name,mode);

int ret;                return code
const char *name;       file name
int mode;               access mode
```

DESCRIPTION

This function checks if a file is accessible in the way specified by mode, which follows the UNIX format:

0	Check if file exists
2	Check if file is writable
4	Check if file is readable
6	Check if file is readable and writable

The other access mode bits recognised by UNIX are not supported under GEMDOS. Also, since all GEMDOS files are readable, modes 0 and 4 are identical, as are modes 2 and 6.

RETURNS

A return value of 0 indicates that access is allowed. If access is denied or the file cannot be found, -1 is returned. Additional error information can then be found in errno and _OSERR.

SEE

chgfa, getfa, errno, _OSERR

Class: UNIX *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

s = alloca(n);

void *s;        pointer to base of memory
size_t n;       number of bytes required
```

DESCRIPTION

The alloca function obtains the specified number of bytes from the program's stack space. The value n gives the number of bytes required, and the return pointer s points to an area of the size requested, or NULL if insufficient stack is available.

Note that you should not attempt to return the space allocated via alloca using the free call. Any space allocated using this function is automatically reclaimed on function exit.

RETURNS

The value s is NULL if no more stack is available.

SEE

calloc, free, malloc, realloc

EXAMPLE

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

FILE *newfile(const char *s)
{
  char *p;

  p=alloca(strlen(s)+5);
  if (!p)
    return NULL;
  strcpy(p,s);
  strcat(p,".tmp");
  return fopen(p,"rb");
}
```

argopt

Get options from argument list

Class: Lattice *Category: Argument Processing*

SYNOPSIS

```
#include <stdlib.h>

optd = argopt(argc,argv,opts,argn,optc);

char *optd;              option data pointer
int argc;                argument count
const char *argv[];      argument vector
const char *opts;        options expecting data
int *argn;               next argument number (changed)
char *optc;              option character (changed)
```

DESCRIPTION

This function examines an argument list to find the next option argument, using conventions similar to those of the UNIX "shell" command processor. These conventions are:

- An option is an argument that begins with a slash (/) or a dash (i.e. a minus sign) and appears between the command verb (i.e. argv(0)) and the first non-option argument. The reason we recognise either a slash or a dash is that the former is an MS-DOS standard, while the latter has been used by UNIX for a long time.

- The character immediately following the dash is called the "option character", and it may be followed by a character string known as the "option data".

- If the option character appears in the opts string, then the data can be separated from the character by white space. In effect, this means that the data might be in the next argv entry if it does not follow the option character in the current entry.

- A dash or slash followed by a blank or a dash indicates the end of the options.

Each time argopt is called, it will find the next option in the argument array and update the integer referenced by argn. On the first call, you should set this integer to 1, since argv(0) points to the command verb. The argc and argv items are normally the same as those passed to your main program, and they are not changed as a result of the argopt calls. The option character is returned in the byte referenced by optc, and the function returns a pointer to the option data string or to a null byte. If the next entry in argv is not an option, then the function returns a NULL pointer.

The opts item provides some flexibility in the way the option data is handled. If opts points to an empty string, then any option data must immediately follow the option character. However, if opts is not empty, then it lists the option characters that always have data. For those characters, the data can be preceded by white space on the command line. What this actually means is that argopt will look at the next entry in argv if the option character is not followed by a data string. If the next entry does not begin with a dash, then it is taken as the option data.

RETURNS

If the next argument is not an option, the function returns a NULL pointer. Otherwise, it returns a pointer to the option data, which will be an empty string if there was no data. If an option was found, the character is placed into the byte referenced by optc, and argn is adjusted to index the next entry in argv.

SEE

getopt, main

EXAMPLE

```
/*
 * Assume that this program is invoked by the
 * following command line:
 *
 *   myprog -x -ypdq -z -g moo -g - blah
 *
 * The output will then be:
 *   Option: x Data:
 *   Option: y Data: pdq
 *   Option: z Data:
 *   Option: g Data: moo
 *   Option: g Data:
 *   Arg[8]: blah
 *
 */

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

char opts[] = "gx";

int main(int argc,char *argv[])
{
  char option,*odata;
  int next;

  for(next = 1;
    odata = argopt(argc,argv,opts,&next,&option); )
    printf("Option: %c, Data: %s\n",option,odata);

  for (; next < argc; next++)
    printf("Arg[%d]: %s\n",next,argv[next]);
  return 0;
}
```

asctime

Generate ASCII time string

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

s = asctime(t);

char *s;                points to time string
const struct tm *t;     points to time structure
```

DESCRIPTION

This function converts a time structure into an ASCII string of *exactly* 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where DDD is the day of the week, MMM is the month, dd is the day of the month, hh:mm:ss is the hour:minute:seconds, and YYYY is the year. For instance:

```
Wed Sep 04 15:13:22 1985\n\0
```

The time pointer returned by the function refers to a static data area that is shared by both ctime or asctime. The time structure argument t is usually returned by the gmtime or localtime function.

SEE

ctime, gmtime, localtime, setlocale

EXAMPLE

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

int main(void)
{
  struct tm *tp;
  time_t t;

  time(&t);
  tp = localtime(&t);
  printf("Current time is %s\n",asctime(tp));
  return 0;
}
```

Class: ANSI *Category: Debugging*

SYNOPSIS

```
#include <assert.h>

assert(exp);

int exp;             expression to be tested
```

DESCRIPTION

The assert macro tests an expression exp for validity (non-zero value). Note that the assert.h header file must be included in your program in order to define the macro. If the expression being tested fails (i.e. is zero) then the program is aborted printing the text of the failing expression, file and line number on stderr.

Also, assert.h contains two versions of the macro. If the symbol NDEBUG is defined, then a null version of the macro is used; otherwise the normal code-generating version applies. This allows you to strip the assertion code from your program without removing the assert calls. To do this, simply define NDEBUG in one of your header files or on the compiler command line via the -d option. In the former case, the header file containing the NDEBUG definition must be included before assert.h.

EXAMPLE

```
/* Make sure integer x is positive */

#include <assert.h>

void postest(int x)
{
  assert(x >= 0);
}
```

Class: ANSI *Category: Process Creation*

SYNOPSIS

```
#include <stdlib.h>

ret = atexit((*func)())

int ret;              0 if successful
void (*func)(void);   function to be registered
```

DESCRIPTION

The atexit function registers the function pointed to by func, to be called without arguments at normal program termination. The atexit function provides a program with a convenient way to clean up the environment before the program exits. It provides a last-in first-out stacking of multiple functions. The chain of registered functions is maintained in such a way that they are invoked in the correct sequence upon program exit.

The functions registered by atexit are invoked before any files are closed or memory is freed. The SIGTERM signal is raised before atexit.

RETURNS

The atexit function returns 0 if the registration succeeds, and non-zero if it fails to allocate memory for its list.

SEE

exit, onexit

Class: ANSI *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

d = atof(p);

double d;         floating point result
const char *p;    input string pointer
```

DESCRIPTION

The atof function converts an ASCII input string into a double value. The string can contain leading white space and a plus or minus sign, followed by a valid floating point number in normal or scientific notation. If scientific notation is used, there can be no white space between the number and the exponent. For example:

```
123.456e-53
```

is a valid number in scientific notation.

EXAMPLE

```
/*
 * This program tests the atof function.
 */

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

int main(void)
{
  char buff[80];
  double d;

  for (;;)
  {
    printf("\nEnter a number: ");
    if(gets(buff) == NULL)
      exit(0);
    if(buff[0] == '\0')
      exit(0);
    d = atof(buff);
    printf("%e\n",d);
  }
  return 0;
}
```

atoi, atol

Convert ASCII to integer

Class: ANSI *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

x = atoi(s);       Convert ASCII to integer
y = atol(s);       Convert ASCII to long integer

int x;             integer result
long int y;        long integer result
const char *s;     input string pointer
```

DESCRIPTION

These functions convert ASCII strings into normal or long integers. The string must have the form:

```
[whitespace][sign]digits
```

where (whitespace) indicates optional leading white space, (sign) indicates an optional + or - sign character, and digits is a continuous string of digit characters. Once the digit portion is reached, the conversion continues until a non-digit character is hit. No check is made for integer overflow.

RETURNS

As noted above.

SEE

atof, stcd_i, stcd_l, strtol, strtoul

_base

Base of stack

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
extern void *_base;
```

DESCRIPTION

This external pointer is used by the stack check code to locate the base of the stack. If the stack pointer is in danger of overrunning this then the function _xcovf is called.

SEE

_STACK, _xcovf

bldmem

Build a memory pool of specified size

Class: OLD *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

bldmem(n);

int n; number of 1K-byte blocks in pool
```

DESCRIPTION

The bldmem function builds up to n contiguous 1K-byte blocks of memory for the pool. If n is 0, the pool is initialised but no memory is allocated.

RETURNS

Returns -1 if memory cannot be allocated.

SEE

getmem, getml, rlsmem, rlsml, sizmem, sbrk

bsearch

Search a data array

Class: ANSI *Category: Search and Sort*

SYNOPSIS

```
#include <stdlib.h>

match=bsearch(key,base,num_mem,size,(*cmp)(obj,arr));

void *match;          matched element or NULL pointer
const void *key;      object to be matched
const void *base;     initial element of searched array
size_t num_mem;       size of array to be searched
size_t size;          size of each element
int (*cmp)();         comparison function
const void *obj;      pointer to key
const void *arr;      pointer to an array element
```

DESCRIPTION

The bsearch function searches an array of num_mem objects (the initial element of which is pointed to by base) for an element that matches the object pointed to by key. The size of each element of the array is specified by size.

The comparison function pointed to by cmp is called with two arguments that point to the key object and to an array element, in that order. The function returns an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array element. The array consists of all the elements that compare less than the key object, all the elements that compare equal to the key object, and all the elements that compare greater than the key object, in that order.

RETURNS

The bsearch function returns a pointer to a matching element of the array, or a NULL pointer if no match is found. If two elements compare as equal, the element matched could be either one.

SEE

lfind, lsearch

_BSSBAS, _DATABAS Base of merged data sections

Class: Lattice *Category: Linker Defined Symbols*

SYNOPSIS

```
extern __far _BSSBAS;
extern __far _DATABAS;
```

DESCRIPTION

These names refer to the base locations in the __MERGED data section. The location of _BSSBAS is the first byte of the merged BSS, whilst _DATABAS is the first byte of the merged data.

SEE

_BSSLEN, _DATALEN

_BSSLEN, _DATALEN

Merged section lengths

Class: Lattice *Category: Linker Defined Symbols*

SYNOPSIS

```
extern __far _BSSLEN;
extern __far _DATALEN;
```

DESCRIPTION

These addresses of these names give the length of the respective __MERGED data section in *longwords.* Note that if you access these variables from assembly language you *must* access them as longs otherwise the assembler may attempt to relocate them, giving random values as a result.

SEE

_BSSBAS, _DATABAS

EXAMPLE

```
/*
 * Clear out the merged BSS in a program
 *
 * Normally done automatically
 */

#include <string.h>

int main(void)
{
  extern __far _BSSBAS;
  extern __far _BSSLEN;

  memset(&_BSSBAS,0,(long)&_BSSLEN/sizeof(long));
  ...
  return 0;
}
```

_bufsiz

Buffered I/O buffer size

Class: Lattice *Category: Stream I/O*

SYNOPSIS

```
extern int _bufsiz;
```

DESCRIPTION

This external integer is used by the buffered I/O system to determine the size of the buffers for buffered files. This location is also used to determine the size of a buffer attached to a file with the setbuf function. In this case, _bufsiz must be set to the size of the buffer before setbuf is called.

Note that the buffer is not allocated when the file is opened. Instead, the first I/O operation causes the buffer to be allocated from the local memory pool if one has not been previously specified with setbuf. This means that if _bufsiz is changed between the open call and the first I/O operation, the size of the buffer allocated for the file will be the value of _bufsiz at the time of the I/O operation, not the value when the file was opened.

SEE

fopen, setbuf, setvbuf

Class: Lattice *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

r = cabs(x);

double r;

struct complex {
  double re;
  double im;
} *x;
```

DESCRIPTION

The cabs function calculates the absolute value of a complex number pointed to by x. cabs(x) returns the value sqrt(x->re * x->re + x->im * x->im). If an overflow occurrs, matherr is called with an OVERFLOW error and suggested return value of HUGE_VAL.

cadd, csub

Complex sum/difference

Class: Lattice *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

z = cadd(x,y,z);
z = csub(x,y,z);

struct complex {
  double re;
  double im;
} *z;

struct complex *x, *y;
```

DESCRIPTION

The cadd function calculates the complex sum of the complex numbers pointed to by x and y, and places the result in the complex number pointed to by z. The pointer z is returned by the function.

Similarly csub calculates the complex difference of the numbers pointed to by x and y, and places the result in the complex number pointed to by z. The pointer z is returned by the function.

For instance, the expression:

```
z = cadd(x,y,z);
```

produces the following assigments:

```
z->re = x->re + y->re;
z->im = x->im + y->im;
```

Whilst the expression:

```
z = csub(x,y,z);
```

produces the following assigments:

```
z->re = x->re - y->re;
z->im = x->im - y->im;
```

calloc

Allocate and clear a memory block

Class: ANSI *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

b = calloc(nelt,esize);

void *b;           block pointer
size_t nelt;       number of elements
size_t esize;      element size
```

DESCRIPTION

The calloc function uses malloc to get a block whose size in bytes is given by:

```
n = nelt * esize;
```

The block is then cleared to zeroes. Like malloc, calloc returns a NULL pointer if the block cannot be allocated.

RETURNS

The calloc function call normally returns a pointer to the block. If there is not enough space for the requested block, or if zero bytes are requested, a NULL pointer is returned.

SEE

free, getmem, malloc, realloc, rlsmem, rbrk, sbrk

Class: Lattice *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

z = cdiv(x,y,z);

struct complex {
  double re;
  double im;
} *z;

struct complex *x, *y;
```

DESCRIPTION

The cdiv function calculates the complex quotient of complex numbers pointed to by x and y, and places the result in the complex number pointed to by z. The pointer z is returned by the function.

For instance, the expression:

```
z = cdiv(x,y,z)
```

produces the assignments:

```
z->re = ( x->re * y->re + x->im * y->im ) /
                (y->re * y->re + y->im * y->im);

z->im = ( x->im * y->re - x->re * y->im ) /
                (y->re * y->re + y->im * y->im);
```

ceil, floor

Get floating point limits

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

x = ceil(y);  Get ceiling of a real number
x = floor(y); Get floor of a real number

double x,y;
```

DESCRIPTION

These functions return the integral values that are nearest to the specified real number. For ceil, the return is the next higher integer, while floor returns the next lower integer.

Note that although these functions return integral values, the results are still real numbers.

EXAMPLE

```
#include <math.h>

double r;

r = ceil(523.96);     /* r contains 524.0 */
r = floor(523.96);    /* r contains 523.0 */
```

cget, cgetc, cgets

Console input operations

Class: Lattice *Category: Console and Port I/O*

SYNOPSIS

```
#include <dos.h>

c = cget();          get character from console,
                     no echo
c = cgetc();         get character from console, echo
p = cgets(buffer);   get string from console

int c;               input character
char *buffer;        input buffer
char *p;             input buffer
```

DESCRIPTION

These functions get single characters or character strings from the console keyboard. The cget and cgetc functions are equivalent to getch and getche, respectively. Also, cgetc and cgets are similar to getchar and gets, respectively. The console functions use the low-level keyboard routines directly rather than working through the file manager. This can result in improved performance in a highly interactive application.

RETURNS

If c is zero, then cget should be called again to obtain the keyboard scan code. This will happen when the user presses a key that cannot be translated into an ASCII code; e.g. a function key. The return from cgets is the buffer pointer.

SEE

cscanf, getch, getche, gets, kbhit

chdir — Change current directory

Class: UNIX *Category: Process Environment*

SYNOPSIS

```
#include <stdio.h>

error = chdir(path);

int error;              0 if successful
const char *path;       points to new directory path
                        string
```

DESCRIPTION

This function changes the current directory to the specified path. Under GEMDOS, the path may begin with a drive letter and a colon.

RETURNS

If the return value is non-zero, then the operation failed. A GEMDOS error code will be in _OSERR, and a UNIX error code will be in errno.

SEE

Dsetpath, mkdir, rmdir, getcd, getcwd

chgclk

Change system clock

Class: Lattice *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

error = chgclk(clock);

int error;
const unsigned char *clock;
```

DESCRIPTION

The chgclk function changes the setting of the system clock, using the following 8-byte array:

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)

RETURNS

If the array is invalid, chgclk returns a non-zero value. In that case, the system clock may be partially changed under GEMDOS, since the date and time are updated on separate GEMDOS calls, either of which may have failed.

If your machine is equipped with a hardware clock, its state is not necessarily changed by a call to chgclk.

SEE

Tsetdate, Tsettime, errno, getclk, _OSERR

chgdsk, getdsk

Change or get current disk drive

Class: GEMDOS *Category: Disk Functions*

SYNOPSIS

```
#include <dos.h>

bmap = chgdsk(drive);
drive = getdsk();

int drive;        drive code
int bmap;         bitmap of mounted drives
```

DESCRIPTION

The chgdsk function changes the current drive code. Drive code 0 corresponds to drive A, code 1 is drive B and so on.

The getdsk function gets the current drive code, using the same codes as chgdsk.

RETURNS

The function chgdsk returns a bitmap of mounted drives, bit 0 corresponds to drive A, bit 1 is drive B and so on.

The function getdsk returns the code of the currently selected drive.

SEE

Dsetdrv, Dgetdrv, getcd

chgdta, getdta

Set/Get data transfer address (DTA)

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

chgdta(dta);
dta = getdta();

struct FILEINFO *dta;    pointer to new DTA
```

DESCRIPTION

The chgdta function is used to change the data transfer address used by GEMDOS in the Fsfirst and Fsnext calls. By comparison the getdta function returns the current data transfer address.

SEE

Fsetdta, Fgetdta, Fsfirst, Fsnext, dfind, dnext

chgfa

Change file attribute

Class: GEMDOS *Category: File System Manipulation*

SYNOPSIS

```
#include <dos.h>

error = chgfa(name,fa);

int error;              0 if successful
int fa;                 file attribute
const char *name;       file name
```

DESCRIPTION

This function sets the attribute byte for the specified file. The attributes in fa are:

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, chmod, getfa, errno, _OSERR

Class: GEMDOS *Category: File System Manipulation*

SYNOPSIS

```
#include <dos.h>

error = chgft(fh,ft);

int error;    0 if successful
long ft;      file time
int fh;       file handle
```

DESCRIPTION

This function sets 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

The chgft function returns 0 if successful or a value of -1 if in error. Additional error information can be found in errno and _OSERR.

SEE

Fdatime, getft, errno, _OSERR

chkml

Check for largest memory block

Class: OLD *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

size = chkml();

long size;
```

DESCRIPTION

This function returns the size, in bytes, of the largest block that is currently available without calling upon the operating system to supply additional heap space.

SEE

getmem, getml, rlsmem, rlsml, sizmem

chkufb
Check unbuffered file handle

Class: Lattice *Category: Low-Level I/O*

SYNOPSIS

```
#include <ios1.h>

ufb = chkufb(fh);

struct UFB *ufb;  pointer to UNIX file block
int fh;           file handle
```

DESCRIPTION

This function checks if a file handle is currently associated with an unbuffered file. Normally it is used internally by open, close, read, write, lseek and tell. The UFB structure is defined in header file ios1.h. For GEMDOS this structure is two short integers. The first contains the mode flags specified in the call to the open function. The second contains the file handle. The external name _ufbs refers to an array of UFB structures, and the external integer _nufbs indicates how many structures are in the array. Normally this value is fourty.

RETURNS

If no UFB is currently attached to the file handle, a NULL pointer is returned.

Class: UNIX *Category: File System Manipulation*

SYNOPSIS

```
#include <stdio.h>

error = chmod(name,mode);

int error;            error code
const char *name;     file name
int mode;             protection mode
```

DESCRIPTION

This function changes a file's protection mode. It is compatible with UNIX, although GEMDOS provides only a single write-protection bit for each file. The mode argument should be formed by ORing any combination of the following symbols which are defined in fcntl.h:

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission

Since all GEMDOS files are readable, only the S_IWRITE symbol actually has any meaning.

RETURNS

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

SEE

access, chgfa, errno, _OSERR

EXAMPLE

```
/*
 * This piece of code changes file "xyz\pdq.x"
 * so it can be read and written.
 */
#include <fcntl.h>

if(chmod("xyz\pdq.x",S_IWRITE | S_IREAD))
  perror("Change mode");
```

clearerr, clrerr

Clear buffered I/O error flag

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

clearerr(fp);
clrerr(fp);

FILE *fp; file pointer
```

DESCRIPTION

The clearerr and clrerr functions clear the error flag associated with the specified file that was previously opened via fopen. Once set, the error flag forces an EOF return any time the file is accessed until the flag is reset.

Note that clearerr is implemented as both a macro and a function. To get the function instead of the macro, include the following line after the #include line:

```
#undef clearerr
```

(The function clrerr is provided for compatibility with some older versions of UNIX.)

SEE

fopen

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

time = clock();

clock_t time; clock time since start of execution
```

DESCRIPTION

The clock function determines the processor time used by the process. The clock is started when the process starts and then clock returns the time elapsed since then.

RETURNS

To determine the time in seconds, the value returned by the clock function should be divided by the value of the macro CLK_TCK. If the processor time used is not available or its value cannot be represented, the function returns the value ((clock_t)-1). This will never be the case under GEMDOS.

EXAMPLE

```
/*
 * time a function, returning a value in seconds
 */

#include <time.h>

long time_me(void (*f)(void))
{
  clock_t start;

  start=clock();
  f();
  return (long)((clock()-start)/CLK_TCK);
}
```

close

Close an unbuffered file

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

error = close(fh);

int error;      non-zero if error
int fh;         file handle
```

DESCRIPTION

This function closes a file that was previously opened via the open function. If there is any pending output, it is completed and the file directory is updated.

All files are automatically closed when your program terminates, but it is good programming practice to close a file when you are finished with it. One reason for doing this is to free up the operating system resources (e.g., control blocks and buffers) that are allocated for the file while it remains open.

RETURNS

The function returns 0 if it is successful. Otherwise, it returns -1 and places additional error information into errno and _OSERR.

SEE

errno, open, _OSERR

EXAMPLE

See the open function.

Class: Lattice *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

z = cmul(x,y,z);

struct complex {
  double re;
  double im;
} *z;

struct complex *x, *y;
```

DESCRIPTION

The cmul function calculates the complex product of complex numbers pointed to by x and y, and place the results in the complex number pointed to by z. The pointer z is returned by the function.

For instance, the expression:

```
z = cmul(x,y,z)
```

produces the following assigment:

```
z->re = (x->re * y->re) - (x->im * y->im);
z->im = (x->re * y->im) + (x->im * y->re);
```

_country

ROM based country identifier

Class: GEMDOS *Category: Process Environment*

SYNOPSIS

```
#include <dos.h>

extern enum {} _country;  country identifier
```

DESCRIPTION

These variable gives the country for which the operating system is nationalised. The currently used values are:

Value	Identifier	Country
0	USA	USA
1	FRG	Germany
2	FRA	France
3	GBR	Great Britain
4	SPA	Spain
5	ITA	Italy
6	SWE	Sweden
7	SWF	Switzerland (French)
8	SWG	Switzerland (German)
9	TUR	Turkey
10	FIN	Finland
11	NOR	Norway
12	DEN	Denmark
13	SAU	Saudi Arabia
14	HOL	Holland

cprintf

Formatted print to console

Class: Lattice *Category: Formatted I/O*

SYNOPSIS

```
#include <conio.h>

length = cprintf(fmt,arg1,arg2,...);

int length;          number of characters generated
const char *fmt;     format string

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 cprintf form of printf sends the stream to the console via a low-level operating system interface, thereby eliminating the buffered I/O overhead.

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

fprintf, lprintf, printf, sprintf, vfprintf, vprintf, vsprintf

cputc, cputs

Console output operations

Class: Lattice *Category: Console and Port I/O*

SYNOPSIS

```
#include <dos.h>

c = cputc(c);           put character to console
count = cputs(buffer);  put string to console

int c;                  input character
int count;              output character count
const char *buffer;     pointer to input string
```

DESCRIPTION

These functions put single characters or character strings to the console display. They are similar to putchar and puts except that they call the low-level video routines instead of working through the File Manager. This can result in better display performance.

RETURNS

The cputc function returns the character that was used as its argument, while cputs returns the number of characters sent to the display.

SEE

cprintf, putchar, puts, kbhit

creat

Create a file

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

fh = creat(name,prot);

int fh;              file handle
const char *name;    file name
int prot;            protection mode
```

DESCRIPTION

This function is exactly the same as calling the open function in the following way:

```
open(name,O_WRONLY | O_TRUNC | O_CREAT |
  (prot & O_RAW),(prot & ~O_RAW));
```

In other words, the file is created if it doesn't exist and truncated if it does exist. Then it is opened for writing, and the translation mode is picked up from the prot argument. The protection mode can be any of the following:

Value	Meaning
S_IWRITE	Write permission
S_IREAD	Read permission
S_IWRITE | S_IREAD	Write and read permission

Also you can OR in O_RAW to suppress file translation. For instance, if prot is

```
O_RAW | S_IREAD
```

the file will be created as read-only and will be processed in raw (untranslated) mode. The read-only condition takes effect only if a new file must be created; if the file already exists, its protection mode is unchanged. Also, you can write to a newly-created read-only file until you close it for the first time.

RETURNS

If the operation succeeds, a file handle is returned, which is a positive integer. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

Fcreate, errno, _OSERR, chgfa, chmod, close, open

cscanf

Formatted input from console

Class: Lattice *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

n = cscanf(fmt,arg1,arg2,...);

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

DESCRIPTION

The cscanf function performs formatted input conversions on text obtained from the system console. 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.

SEE

fscanf, scanf, sscanf

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

s = ctime(t);

char *s;            points to time string
const time_t *t;    points to time value
```

DESCRIPTION

This function converts a Greenwich Mean Time (GMT) time value to an ASCII string of *exactly* 26 characters having the form:

```
DDD MMM dd hh:mm:ss YYYY\n\0
```

where DDD is the day of the week, MMM is the month, dd is the day of the month, hh:mm:ss is the hour:minute:seconds, and YYYY is the year. For instance:

```
Wed Sep 04 15:13:22 1985\n\0
```

The time pointer returned by the function refers to a static data area that is shared by both ctime and asctime.

The time value argument t must point to a long integer that is the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. Normally this value is obtained from the time function. Note that ctime converts this value back into local time by calling _tzset and then subtracting the contents of timezone.

Note that t is a pointer to a time_t. A common error is to pass the time_t value itself instead of the pointer. Observe the use of the ampersand (&) operator in the following example.

SEE

asctime, gmtime, localtime, time, _tzset, utpack, utunpk

EXAMPLE

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

int  main(void)
{
  time_t  t;

  time(&t);
  printf("Current  time  is  %s\n",ctime(&t));
}
```

_CXFERR

Low-level float error exit

Class: Lattice *Category: Errors*

SYNOPSIS

```
#include <math.h>

_CXFERR(code);

int code;
```

DESCRIPTION

The _CXFERR function is called when an error is detected by one of the low-level floating point routines, such as arithmetic operations. Higher-level routines, such as trigonometric functions, use the more sophisticated matherr.

Users can replace this error trap with an application-dependent routine, as long as they still store the error code in the global integer _FPERR. This is necessary because some of the maths functions check _FPERR to see if low-level errors occurred.

The error code passed to _CXFERR indicates the type of floating point anomaly that occurred, as follows, defined in math.h:

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

SEE

matherr

_dclose

Close a GEMDOS file

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include  <dos.h>

error  =  _dclose(fh);

long  error;    0 for success, -1 for error
int  fh;        file handle
```

DESCRIPTION

This function closes a GEMDOS file that was opened via _dcreat, _dcreatx or _dopen.

RETURNS

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

SEE

Fclose, errno, _OSERR, _dcreat, _dcreatx, _dopen

_dcreat, _dcreatx — Create a GEMDOS file

Class: GEMDOS — *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

fh = _dcreat(name,fatt);   Create or truncate GEMDOS
                           file
fh = _dcreatx(name,fatt);  Create new GEMDOS file

long fh;                   file handle (-1 for error)
const char *name;          file name
int fatt;                  file attribute
```

DESCRIPTION

These functions create and open a GEMDOS file, returning the file handle. The _dcreat operation will truncate the file if it already exists, or create the file if it does not exist. Alternatively, _dcreatx will fail if the file already exists.

RETURNS

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

SEE

Fcreate, errno, _OSERR, _dopen

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include  <dos.h>

err  =  dfind(info,name,attr);  Find  first  directory
                                entry
err  =  dnext(info);            Find  next  directory
                                entry

int  err;                       0  if  successful
struct  FILEINFO  *info;        file  information  area
const  char  *name;             file  name  or  pattern
int  attr;                      file  attribute  bits
```

DESCRIPTION

These functions search a directory for entries that match the specified file name or file name pattern. The dfind function locates the first matching file. Then successive calls to dnext locate additional matching files. Each dnext call must be given the file information that was returned on the preceding call to dfind or dnext.

The name argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case the current directory will be searched. You can use the GEMDOS * and ? characters for pattern matching in the name portion. For example, xy*.b will locate files in the current directory that begin with xy and have b as their extension.

The attr argument specifies which file types are to be included in the search. The following bits are used:

Bit	Meaning
0	Read-only flag
1	Hidden file flag
2	System file flag
3	Volume label flag
4	Subdirectory flag

The info argument points to a file information structure as defined in the dos.h header file. For GEMDOS, this is the same as the GEMDOS DTA structure:

```
struct  FILEINFO
{
   char  resv[21];        /* reserved */
   char  attr;            /* actual file attribute */
   long  time;            /* file time and date */
   long  size;            /* file size in bytes */
   char  name[FNSIZE];    /* file name */
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in errno and _OSERR.

SEE

Fsfirst, Fsnext, getfnl, errno, _OSERR

EXAMPLE

```
/*
 * show the files in a given directory
 */

#include <dos.h>

void showdir(const char *s)
{
  struct FILEINFO info;

  if (!dfind(&info,name,0)
    do
    {
      puts(info.name);
    } while (!dnext(&info));
}
```

difftime — Compute difference between calendar times

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

diff = difftime(time1,time0);

double diff;      difference between calendar times
                  (seconds)
time_t time1;     one calendar time
time_t time0;     another calendar time
```

DESCRIPTION

The difftime function computes the difference (in seconds) between two calendar times: time1 - time0. difftime was introduced as an ANSI function so that implementations could store an indication of the date/time value in the most efficient format possible and still provide a method of calculating the difference between two times.

RETURNS

This function returns the difference expressed in seconds as a double.

_disatty

Check if a GEMDOS handle is a terminal

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include  <dos.h>

ret  =  _disatty(fh);

int  ret;    0 if  not  a  terminal
int  fh;     file  handle
```

DESCRIPTION

This function returns a non-zero value if the specified GEMDOS file handle is attached to a terminal (TTY) device, i.e. a console, printer or auxiliary device.

RETURNS

The return value is 0 if the file is not a terminal or if an error occurred while attempting to obtain the file's characteristics. You can check errno and _OSERR for detailed error information. If the file is a terminal, a value of 1 is returned.

SEE

isatty, errno, _OSERR

div, ldiv

Divide two signed integers

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

p = div(numer,denom);     Divide two signed integers
q = ldiv(lnumer,ldenom)   Divide two signed longs

div_t p;                  quotient, remainder
ldiv_t q;                 long quotient, remainder
int numer;                numerator
int denom;                denominator
long lnumer;              long numerator
long ldenom;              long denominator
```

DESCRIPTION

The div and ldiv functions compute the quotient and remainder of the division of the numerator by the denominator. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. The result can be represented as:

```
p.quot * denom + p.rem = numer
```

The div and ldiv functions provide a set of well-specified semantics for signed integral division and remainder operations. The semantics were adopted to be the same as FORTRAN. The following table summarises the semantics of these functions:

Numerator	Denominator	Quotient	Remainder
7	3	2	1
-7	3	-2	-1
7	-3	-2	1
-7	-3	2	-1

RETURNS

The div function returns a structure of type div_t, comprising both the quotient and the remainder, whilst the ldiv function returns a structure of type ldiv_t. The structures contain the following members:

```
int quot;    /* quotient */
int rem;     /* remainder */
```

_dopen — Open a GEMDOS file

Class: GEMDOS — *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

fh = _dopen(name,mode);

long fh;                    file handle (-1 for error)
const char *name;           file name
int mode;                   access mode
```

DESCRIPTION

This function opens a GEMDOS file and returns the file handle. The mode argument must be a mode supported directly by GEMDOS, i.e. O_RDONLY, O_WRONLY and O_RDWR.

RETURNS

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

SEE

Fopen, errno, _OSERR, open, _dcreat, _dcreatx, _dclose

Class: UNIX — *Category: Random Numbers*

SYNOPSIS

```
#include <math.h>

x = drand48();              random double (internal seed)
x = erand48(seed);          random double (external seed)
y = lrand48();              random positive long (internal
                            seed)
y = nrand48(seed);          random positive long (external
                            seed)
z = mrand48();              random long (internal seed)
z = jrand48(seed);          random long (external seed)
srand48(hseed);             set high 32 bits of internal
                            seed
pseed = seed48(seed);       set all 48 bits of internal
                            seed
lcong48(parm);              set linear congruence
                            parameters

double x;                   random double
long y;                     random positive long
long z;                     random long
short seed[3];              seed value (high bits in
                            seed[0])
long hseed;                 high 32 bits of seed value
short *pseed;               pointer to internal seed
short parm[7];              parameters
```

DESCRIPTION

These functions generate various types of random numbers using the linear congruential algorithm and 48-bit arithmetic. The normal functions drand48, lrand48 and mrand48 use an internal 48-bit storage area for the seed value. Special versions erand48, jrand48 and nrand48 are provided for cases where several seeds are in use at the same time, in which case the user specifies the seed on each function call.

The drand48 and erand48 functions return double values distributed uniformly over the interval from 0.0 up to but not including 1.0.

The lrand48 and nrand48 functions return non-negative long integers uniformly distributed over the interval from 0 to 2**31-1.

The mrand48 and jrand48 functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The srand48 and seed48 functions allow initialisation of the internal 48-bit seed to something other than the default. For srand48 the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to 0x330E. For seed48 the entire 48 bits are loaded from the specified array, and the function returns a pointer to the internal seed array.

The lcong48 function allows a much more intricate initialisation of the linear congruential algorithm. The algorithm is of the form:

```
X[n+1] = (a * X[n] + c) mod m
```

where m is 2**48 and the default values for a and c are 0x5DEECE66D and 0xB, respectively. The array passed to lcong48 is structured as follows:

Parameter	Value
parm(0)	Bits 47-32 of value X(n)
parm(1)	Bits 31-16 of value X(n)
parm(2)	Bits 15-00 of value X(n)
parm(3)	Bits 47-32 of value a
parm(4)	Bits 31-16 of value a
parm(5)	Bits 15-00 of value a
parm(6)	value c

Whenever seed48 is called, a and c are reset to their default values.

RETURNS

As noted above.

SEE

rand, srand

_dread, _dwrite

Read and write GEMDOS files

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

cnt = _dread(fh,buf,len);    Read from a GEMDOS file
cnt = _dwrite(fh,cbuf,len);  Write to a GEMDOS file

long cnt;                    actual bytes read or
                             written
int fh;                      file handle
void *buf;                   data buffer
const void *cbuf;            data buffer
size_t len;                  number of bytes to read
                             or write
```

DESCRIPTION

These functions read or write a GEMDOS file whose handle was returned by _dcreat, _dcreatx or _dopen. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult errno and _OSERR. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check errno and _OSERR just in case some malfunction caused the short count.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

errno, _OSERR, _dcreat, _dcreatx, _dopen, _dclose, _dseek

_dseek

Re-position a GEMDOS file

Class: GEMDOS *Category: DOS Interface*

SYNOPSIS

```
#include  <dos.h>

apos  =  _dseek(fh,rpos,mode);

long  apos;      actual  file  position
int  fh;         file  handle
long  rpos;      relative  file  position
int  mode;       seek  mode
```

DESCRIPTION

This function re-positions a GEMDOS file whose handle was returned by _dcreat, _dcreatx or _dopen. The seek mode is the same as for lseek as follows (defined in stdio.h):

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.

Note that for mode SEEK_CUR rpos can be positive or negative, but apos is always the actual (positive) position relative to the beginning of file.

RETURNS

If the operation is successful, the function returns the actual file position, which is a long integer. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

Fseek, errno, _OSERR, _dread, _dwrite

_ddup, _ddup2 — Duplicate a GEMDOS file handle

Class: GEMDOS — *Category: DOS Interface*

SYNOPSIS

```
#include <dos.h>

nfh = _ddup(fh);                Duplicate a file handle
error = _ddup2(nfh,fh);         Assign a file handle

int nfh;                        new file handle
int fh;                         old file handle
int error;                      -1 if error
```

DESCRIPTION

These functions duplicate a GEMDOS file handle. The new handle is associated with the same file as the old handle.

They are normally used in the same way as the higher level dup and dup2 functions for associating a different stdin, stdout, or stderr for a child process.

RETURNS

If the operation is successful, _ddup returns a file handle, while _ddup2 returns 0. Otherwise a value of -1 is returned, and error information is placed into errno and _OSERR.

Do not use these functions with files being accessed via open and the other low-level I/O functions. Use dup and dup2 instead.

SEE

Fdup, Fforce, dup, dup2, _dopen, _dclose, errno, _OSERR

dup, dup2

Duplicate a file handle

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

nfh = dup(fh);          Duplicate a file handle
error = dup2(nfh,fh);   Assign a file handle

int nfh;                new file handle
int fh;                 old file handle
int error;              -1 if error
```

DESCRIPTION

These functions duplicate a file handle. The new handle is associated with the same file as the old handle.

Normally, dup is used when you want to establish a different stdin, stdout, or stderr for a child process. In order to preserve your current input, output, or error channel, you would use either dup or dup2 to duplicate file handle 0, 1, or 2. Then you would use fdopen to re-establish the association between the new handle and stdin, stdout, or stderr. Finally, you would open a file that you want to be the child process' standard input, output, or error channel; use dup2 if necessary to make the proper association with handle 0, 1, or 2.

RETURNS

If the operation is successful, dup returns a file handle, while dup2 returns 0. Otherwise a value of -1 is returned, and error information is placed into errno and _OSERR.

Do not use these functions with files being accessed via _dopen and the other low-level I/O functions. Use _ddup and _ddup2 instead.

SEE

Fdup, Fforce, _ddup, _ddup2, fdopen, errno, _OSERR

ecvt, fcvt

Convert float to string

Class: UNIX *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <math.h>

s = ecvt(v,dig,decx,sign);  convert float to string
s = fcvt(v,dec,decx,sign);  convert float to string

char *s;                    string pointer
double v;                   floating point value
int dig;                    number of digits
int dec;                    number of decimal places
int *decx;                  pointer to decimal index
                            (returned)
int *sign;                  pointer to sign indicator
```

DESCRIPTION

These functions convert a floating point number into an ASCII character string consisting of digits only and terminated by a null character.

For ecvt, the second argument indicates the total number of digits that should be generated, while for fcvt it indicates how many digits should be generated to the right of the decimal place. If the floating point value contains fewer significant digits, zeroes are appended. If there are too many significant digits, the low order (right-most) digit is rounded.

The decx argument points to an integer that will receive a value indicating where the decimal point should be placed in the string. For example, an index value of 3 indicates that the decimal point should be placed just after the third character in the string. A value of zero means that the decimal point is just before the first character. If the index is negative, it indicates the number of zeroes that are between the decimal point and the first character. For example, -3 means that there are three zeroes between the decimal point and the beginning of the string.

The sign argument points to an integer that will be non-zero if v is negative.

EXAMPLE

```
#include <math.h>

int main(void)
{
  int decx,sign;
  char *string;

  string = ecvt(3.1415926535,10,&decx,&sign);
```

```
/*
 *  string  =>  "3141592654"
 *  decx  =>  1
 *  sign  =>  0
 */

  string  =  fcvt(3.1415926535,10,&decx,&sign);

/*
 *  string  =>  "31415926535"
 *  decx  =>  1
 *  sign  =>  0
 */
  return  0;
}
```

__emit

Emit 68000 instruction word

Class: Lattice *Category: Builtin Functions*

SYNOPSIS

```
#include  <dos.h>

__emit  (x);

short x;   opcode to place in instruction stream
```

DESCRIPTION

The built-in function emit takes a constant 16-bit value corresponding to a 68000 assembly language instruction and inserts it in-line with the code. However, it does not check whether the 16-bit value is a valid 68000 instruction. It lacks the power and flexibility of an in-line assembler.

Note that this function is implemented as a macro expanding to the function __builtin_emit hence you *must* include the header file dos.h.

If one doesn't know how to use the emit function, it can create serious problems. While programmers may find this function useful in some situations, it should not be used without exercising a great deal of care and skill.

SEE

getreg, putreg

_end, _edata, _etext Last locations in program

Class: UNIX *Category: Linker Defined Symbols*

SYNOPSIS

```
extern __far _end;
extern __far _data;
extern __far _etext;
```

DESCRIPTION

These names refer to the last locations in the program. The address of _etext is the first location above the executable program text, that of _edata the first location above the initialised data area and _end the location immediately after the unitialised data area.

_ENEED

Maximum environment string space

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
extern int _ENEED;
```

DESCRIPTION

This external variable specifies the maximum number of environment strings which may be manipulated by the getenv, putenv and rmvenv commands. If it is smaller than that required for the process when it starts the value is ignored and the value allocated 4 times the number of strings available at startup.

environ

Strings forming user environment

Class: UNIX *Category: Process Environment*

SYNOPSIS

```
extern char **environ;
```

DESCRIPTION

The external variable environ points to an array of strings forming the "environment". By convention these strings have the form "NAME=value". This array is normally manipulated by the functions getenv, putenv and rmvenv.

SEE

getenv, putenv, rmvenv, _ENEED

Class: ANSI *Category: Errors*

SYNOPSIS

```
#include <errno.h>

extern int volatile errno;    UNIX error number
extern int sys_nerr;          number of error codes
extern char *sys_errlist[];   UNIX error messages
```

DESCRIPTION

The external integer named errno is initialised to 0 at start-up time. Then if an error is detected by one of the standard library functions, a non-zero value is placed there. The standard library never resets errno.

Programmers typically use this information in two ways. In some cases, it is appropriate to check errno after a sequence of operations and abort if any error occurred along the way. In other cases, errno is checked periodically, and if it is non-zero, the appropriate corrective action is taken. Then the application program resets errno before beginning the next processing phase.

The sys_nerr and sys_errlist items are defined in a C source file named syserr.c and are used by the perror function to print messages that correspond to the code found in errno. Note that the sys_ variables do *not* form part of the ANSI C standard.

Note that even though error information is normally placed into errno by the standard library functions, application programs can also use this technique to indicate problems. However, you should be careful about adding new codes and messages just above the highest UNIX code currently defined, since new UNIX codes are added occasionally. Also, we recommend that you add application-dependent codes by extending the header file errno.h, which contains symbolic definitions of the code numbers. The currently defined codes are listed as follows:

Symbol	Code	Meaning
EOSERR	-1	Operating system error
EPERM	01	User is not owner
ENOENT	02	No such file or directory
ESRCH	03	No such process
EINTR	04	Interrupted system call

EIO	05	I/O error
ENXIO	06	No such device or address
E2BIG	07	Argument list is too long
ENOEXEC	08	Exec format error
EBADF	09	Bad file number
ECHILD	10	No child process
EAGAIN	11	No more processes allowed
ENOMEM	12	No memory available
EACCES	13	Access denied
EFAULT	14	Bad address
ENOTBLK	15	Bulk device required
EBUSY	16	Resource is busy
EEXIST	17	File already exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Is not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	No more files (system)
EMFILE	24	No more files (process)
ENOTTY	25	Not a terminal
ETXTBSY	26	Text file is busy
EFBIG	27	File is too large
ENOSPC	28	No space left
ESPIPE	29	Seek issued to pipe
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math function argument error
ERANGE	34	Math function result is out of range

SEE

perror, strerror, sys_err

exit, _exit

Terminate program execution

Class: ANSI *Category: Process Creation*

SYNOPSIS

```
#include <stdlib.h>

exit(code);   Terminate with clean-up
_exit(code);  Terminate with no clean-up

int code;     status code
```

DESCRIPTION

These functions terminate execution of the current program and return control to the parent program. Use exit, for a graceful termination, which means that all pending output buffers are written and all files are explicitly closed. The _exit function terminates immediately without writing output buffers or closing files. Generally, this latter form is used only in emergency situations when you don't care if some output data is lost.

This function will normally be called after the code in main has been executed, and any return value from main is then passed to exit. Note that in general the _exit function is automatically called from the exit function after it has performed any clean up required.

In either case, the code is a value that gets passed back to the parent. By convention, a value of zero indicates success. If the parent is another C program that started this one up via one of the fork functions, then the parent can obtain the return code via the wait function.

RETURNS

This function does not return.

SEE

Pterm, Pterm0, onexit, atexit, forklpe, forkvpe, wait

EXAMPLE

```
/*
 * This example shows how you would abort a program
 * if it is not called with a valid input file name.
 *
 */
#include <stdio.h>
#include <stdlib.h>

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

  if(argc > 1)
  {
    f = fopen(argv[1],"r");
    if(!f)
    {
      fprintf(stderr,"Can't open file %s\n",argv[1]);
      return 1;
    }
  }
  else
  {
    fprintf(stderr,"No file specified\n");
    return 1;
  }

/*** Continue, now that file has been verified ***/
```

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

r = exp(x);         exponential function
r = log(x);         natural logarithm function
r = log10(x);       base 10 logarithm function
r = pow(x,y);       power function
r = sqrt(x);        square root function
r = pow2(x);        compute 2**x

double r, x, y;
```

DESCRIPTION

The exp function raises the natural logarithm base e to the x power, and pow raises x to the y power. For pow, the x value must be an integer if it is negative. If it is not integral, matherr is called with a DOMAIN error.

The pow2 function computes 2^x by calling the pow function. The return value r is the value 2^x .

The log and log10 functions take the base e and base 10 logarithm, respectively. Each of these as well as sqrt, requires a positive argument. If a negative argument is supplied, matherr will be called with a DOMAIN error.

SEE

matherr

fabs

Absolute value of float/double

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <math.h>

ad = fabs(d);

double d;
double ad;
```

DESCRIPTION

The fabs function computes the absolute value of a float or a double, returning a double result.

SEE

abs, iabs, labs

fclose, fcloseall

Close a buffered file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

ret = fclose(fp);   close a buffered file
num = fcloseall();  close all buffered files

int ret;            return code
int num;            number of files closed
FILE *fp;           file pointer for file to be
                    closed
```

DESCRIPTION

The fclose function completes the processing of a buffered file (i.e. a file previously opened via fopen) and releases all related resources. The buffer associated with the file is released via the free function.

Even though fclose is automatically called for all open files when your program terminates or calls exit, it is good programming practice to close your own files explicity. The the last buffer is not written until fclose is called, and so data may be lost if an output file is not properly closed.

The fcloseall function closes all buffered files and returns the number of files that were closed. If an error occurs on any file, fcloseall continues to close the other files and then returns a value of -1.

RETURNS

Both functions return -1 to indicate an error. For success, fclose returns 0, and fcloseall returns the number of files that were closed. If -1 is returned, additional error information can be found in errno and _OSERR.

Remember that fcloseall closes the standard files stdin, stdout, and stderr. This means, for example, that functions such as printf and perror will fail after you call fcloseall.

SEE

fopen, errno, _OSERR

Class: UNIX *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

fp = fdopen(fh,mode);

FILE *fp;            file pointer
int fh;              file handle
const char *mode;    access mode
```

DESCRIPTION

This function assigns a specific file handle to a buffered file. In other words, if you have used open to obtain a file handle, you can subsequently use buffered I/O with that file via fdopen. The mode argument for fdopen has the same form as for fopen.

RETURNS

If the operation is successful, the function returns a non-NULL file pointer. Otherwise it returns a NULL pointer and places error information in errno and _OSERR.

SEE

fopen, errno, _OSERR

feof

Check for end-of-file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

ret = feof(fp);

int ret;  non-zero if end-of-file is found
FILE *fp; file pointer
```

DESCRIPTION

The feof function generates a non-zero value if the specified file is at end-of-file. Note that the specified file must have been opened previously via fopen or fdopen.

RETURNS

If an end-of-file is found, a non-zero value is returned.

This function is implemented as a macro, and does not check if fp is a valid file pointer.

SEE

ferror

ferror

Check for file error

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

ret = ferror(fp);

int ret;      non-zero if file error is found
FILE *fp;     file pointer
```

DESCRIPTION

The ferror function generates a non-zero value if an error has occurred on the specified file. Note that the file must have been opened previously via fopen or fdopen.

RETURNS

The return value is 0 if no error has occurred. If a file error has been found, a non-zero value is returned.

The ferror function is implemented as a macro, and does not check if fp is a valid file pointer.

SEE

feof

fflush, flushall

Flush file output buffer

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

ret = fflush(fp);  Flush a file output buffer
num = flushall();  Flush all file output buffers

FILE *fp;          file pointer
int ret;           return code
int num;           number of open files
```

DESCRIPTION

The fflush macro flushes the output buffer of a file previously opened via fopen or fdopen. That is, it writes the buffer if the file is opened for output and the buffer contains any pending data. If an error occurs, the return value is EOF and the appropriate error code is placed into errno.

The flushall function flushes all file output buffers and returns the number of files that are open. If an error occurs, the function continues to flush the remaining files and then returns a value of -1.

RETURNS

As noted above. In the event of a -1 return, error information can be found in errno and _OSERR.

SEE

fopen, fclose, errno, _OSERR

fgetc, fgetchar

Get a character

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

c = fgetc(fp);     Get a character from a file
c = fgetchar();    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 fgetchar, the standard input file is read via file pointer stdin.

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 errors from end-of-files, you should reset errno before calling the function and then analyse its contents when you receive an EOF return.

SEE

errno, fopen, getc, getchar, _OSERR

fgetpos

Store current value of file position indicator

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include  <stdio.h>

ret  =  fgetpos  (strm,pos);

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

DESCRIPTION

The fgetpos function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored in pos contains information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function.

RETURNS

If successful, the fgetpos function returns 0; on failure, the fgetpos function returns non-zero and stores an the error value in errno.

SEE

fsetpos

fgets

Get a string from a buffered file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

p = fgets(buffer,length,fp);

char *p;        buffer pointer or NULL
char *buffer;   buffer pointer
int length;     buffer length in bytes
FILE *fp;       file pointer
```

DESCRIPTION

The fgets function gets a string from the specified file, which must have been previously opened for input via fopen or fdopen. Characters are copied from the file to the buffer until a newline ('\n') has been copied, or the buffer is full, or the end-of-file is hit. In the newline case, a null byte ('\0') is placed into the buffer after the newline if the buffer has room. In the end-of-file case, a null byte is placed into the buffer after the last byte that was read. If the end-of-file is hit before any bytes are read, a NULL pointer is returned.

Note that the returned string will not be null-terminated if length characters have already been placed into the buffer.

RETURNS

The fgets function returns the buffer argument unless an end-of-file or I/O error occurs, in which case a NULL pointer is returned.

SEE

errno, feof, ferror, fgetc, fopen, getc, gets

EXAMPLE

```
/*
 * Assume that stdin contains the following lines:
 *
 * Hello, folks!
 * Goodbye, folks!
 * (blank line or EOF)
 */
#include <stdio.h>

  char *p,b[80];
  /* For the next two lines, p will point to b */
  p = gets(b);
  /* Now b contains "Hello, folks!" */
  p = fgets(b,sizeof(b),stdio);
  /* Now b contains "Goodbye, folks!\n" */
  p = gets(b);
  /* Now p is NULL */
```

fgetw, fgetl

Get a word/longword from a buffered file

Class: UNIX *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

x = fgetw(fp);
y = fgetl(fp);

short x;        word value from stream
long y;         longword value from stream
FILE *fp;       file pointer
```

DESCRIPTION

The fgetw and fgetl functions read words and longwords respectively from the associated file. If end-of-file is reached, 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 should be checked in these cases.

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

RETURNS

The functions return a value from the stream or the value EOF if an end-of-file or I/O error occurs.

SEE

errno, feof, ferror, fgetc, fread, fputw, fputl

filelength

Find length of an unbuffered file

Class: Microsoft *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

length = filelength(fh);

long length;   length of file in bytes or -1
int fh;        unbuffered file handle
```

DESCRIPTION

The filelength function calculates the size of the file associated with the unbuffered file handle fh. The file handle should be one which was returned by an open or creat call.

RETURNS

The filelength function returns the number of bytes in the file, or if an error occurs returns -1 and sets errno accordingly.

SEE

creat, fileno, open

EXAMPLE

```
/*
 * Find the length of a buffered file
 */
#include <stdio.h>
#include <fcntl.h>

long len(FILE *fp)
{
  fflush(fp); /* flush any buffered bytes to disk */
  return filelength(fileno(fp));
}
```

fileno

Get handle for buffered file

Class: UNIX *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

fh = fileno(fp);

int fh;     file handle
FILE *fp;   file pointer
```

DESCRIPTION

This function returns the file handle (i.e. the file number) associated with the specified file pointer. The file pointer must be one that was returned by fopen, freopen, or fdopen.

RETURNS

As noted above.

This function is implemented as a macro, and it does not check that fp is a valid file pointer.

_fmask

Set default protection mode for buffered I/O

Class: Lattice *Category: Stream I/O*

SYNOPSIS

```
extern long _fmask;
```

DESCRIPTION

This external integer is used by the fopen function to determine the protection mode to use when creating buffered files. The default is the value S_IWRITE | S_IREAD, giving both read and write privileges to any file created.

SEE

fopen

fmod

Compute floating point modulus

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <math.h>

x = fmod(y,z);

double x;        floating point modulus
double y;        dividend
double z;        divisor
```

DESCRIPTION

The fmod function computes the floating point remainder of y/z. It returns y if z is 0. Otherwise, it returns a value that has the same sign as y, is less than z, and satisfies the relationship:

```
y = (i * z) + x
```

where I is an integer. This is, in effect, what the expression:

```
x = y % z;
```

would produce if the % operator were defined for floating point numbers.

SEE

modf

EXAMPLE

```
#include <math.h>

double r,ff,fi;

r = fmod(5.7,1.5);     /* r contains 1.2 */

ff = modf(r,&fi);      /* ff contains 0.2 */
                       /* fi contains 1.0 */
```

_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

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
```

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

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

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

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

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

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

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

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));
}
```

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);
```

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

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.

Class: Lattice *Category: Builtin Functions*

SYNOPSIS

```
#include <dos.h>

value = getreg(reg);        obtain value of a register
putreg(reg,value);          set up the a register

int reg;                    number of register to use
long value;                 value to get/set
```

DESCRIPTION

The built-in function getreg takes as its parameter a constant integer in the range of 0 to 15. The number that you pass is the register number for which you want the current contents. Numbers 0 to 7 correspond to the D0-D7 registers, while numbers 8 to 15 correspond to the A0-A7 registers. The macros REG_D0 to REG_A7 are provided to give names to these numbers in the dos.h header file.

The built-in function putreg takes as its parameter the register number as described above for getreg. The number that you pass is a long integer, which is placed in the specified register.

Incorrect use of these functions can cause serious problems. These functions are intended for use with interrupt code. For instance, the getreg function is useful for obtaining the value of the system registers (e.g. A4) to be passed to an interrupt chain. However, the getreg function is not a reliable way of getting the value of a variable because the code generator may change code generation style during compile time. While programmers may find these functions useful in some situations, a great deal of care and skill should be exercised in their use.

RETURNS

The getreg function returns the current value of the register (a long integer). The putreg function does not return a value.

gets — Get a string from stdin

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

p = gets(buffer);

char *p;       buffer pointer or NULL
char *buffer;  buffer pointer
```

DESCRIPTION

The gets function copies characters from the standard input file, stdin, until a newline is reached. The newline is not copied to the buffer, but a null byte ('\0') is put there in its place.

See the description of the fgets function for an example of the use of both fgets and gets.

Make sure that your gets buffer can hold the largest line that will be encountered while reading stdin, because the function does not have any way to check for a maximum length.

RETURNS

The gets function returns the buffer argument unless an end-of-file or I/O error occurs, in which case a NULL pointer is returned.

SEE

errno, feof, ferror, fgetc, fgets, fopen, getc

gmtime

Unpack Greenwich Mean Time

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

ut = gmtime(t);

struct tm *ut;
const time_t *t;
```

DESCRIPTION

The gmtime function unpacks a time value from the time_t form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. The time function (described elsewhere) returns this kind of number. For gmtime, this number is converted "as is", without any adjustment for the local time zone.

Note that the gmtime function expects a pointer as the argument. A common error is to pass the actual time value instead of the pointer.

Also, localtime and gmtime share a static data area for their return values. A call to either one will destroy the results of the previous call.

SEE

asctime, ctime, localtime, time, _tzset, utpack, utunpk

EXAMPLE

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

int main(void)
{
  struct tm *p;
  time_t t;

  time(&t);
  p = gmtime(&t);
  printf("GMT is %s\n",asctime(p));
}
```

_hash

Compute hash value

Class: Lattice *Category: String Search*

SYNOPSIS

```
#include <stdlib.h>

x= _hash(s);

size_t x;          hash value of string
const char *s;     string to obtain hash value for
```

DESCRIPTION

The _hash function computes a hashing function based on all characters in the string s. The function used is extremely fast and gives an excellent distribution for all strings. It is based on P. J. Weinberger's algorithm and can be found in "Compilers: Principles, Techniques and Tools", see the Bibliography.

SEE

bsearch, lsearch

EXAMPLE

```
/*
 * maintain a hash table, given an item insert
 * it if not found, else return a pointer to it
 */

#include <stdlib.h>

#define HASHMAX 211   /* prime number */

typedef struct hash
{
  struct hash *next;
  char *s;
} hash_t;

struct hash_t hashtab[HASHMAX];

hash_t *lookup(const char *s)
{
  hash_t *p;

  /* find initial element */
  p=&hashtab[_hash(s)%HASHMAX];

  /*
   *walk list until we have a match or the list is
   * empty
  */
  while (*p && strcmp((*p)->s,s))
    p=&(*p)->next;
```

```
  /* if not found then insert */
  if (!*p)
  {
    /* get more memory and insert it into list */
    *p=malloc(sizeof(hash_t));
    (*p)->next=NULL;
    (*p)->next=s;
  }
  return *p;
}
```

iabs

Integer absolute value

Class: Lattice *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

as = iabs(s);

int s;          integer value
int as;         absolute value of s
```

DESCRIPTION

The labs function computes the absolute value of an integer. The abs has the same purpose.

SEE

abs, fabs, labs

_iomode

Default unbuffered I/O mode

Class: Lattice *Category: Low-Level I/O*

SYNOPSIS

```
extern int _iomode;
```

DESCRIPTION

This external integer is used by the open function to determine the translation mode to use when the programmer does not specify a mode in the open 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

open

iomode

Change mode of unbuffered file

Class: Lattice *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

error = iomode(fh,mode);

int error;      error code
int fh;         file handle
int mode;       0 => translated mode
                1 => raw mode
```

DESCRIPTION

This function changes the mode of an unbuffered file whose handle was previously returned by open.

When in translated mode, carriage returns are deleted on input, and a carriage return is inserted before each line feed on output. In raw mode, all data in the file is transferred as is.

Note that iomode affects only the software translation that is done by the library functions.

RETURNS

A non-zero return value indicates that the specified file handle is not valid. That is, it was not returned by open.

SEE

open

Class: ANSI *Category: Character Classification/Conversion*

SYNOPSIS

```
#include <ctype.h>

t = isalnum(c);       Test if alphanumeric character
t = isalpha(c);       Test if alphabetic character
t = isascii(c);       Test if ASCII character
t = iscntrl(c);       Test if control character
t = iscsym(c);        Test if C symbol character
t = iscsymf(c);       Test if C symbol lead character
t = isdigit(c);       Test if decimal digit character
t = isgraph(c);       Test if graphic character
t = islower(c);       Test if lower case character
t = isprint(c);       Test if printable character
t = ispunct(c);       Test if punctuation character
t = isspace(c);       Test if space character
t = isupper(c);       Test if upper case character
t = isxdigit(c);      Test if hex digit character

int t;                truth value   0 => false
                                    non-zero => true
int c;                character to test
```

DESCRIPTION

These functions test for various character types. If you include ctype.h as shown above, then the functions are actually defined as macros and generate in-line code to test the static array named _ctype. This array contains a bit mask for each of the 256 possible character values and for the integer value -1. See the ctype.h for the bit definitions.

If you don't include ctype.h, these functions will be included from the library, which can reduce your program size slightly at the expense of execution speed. If you want to use the function versions but must include ctype.h for some other reason, use #undef to undefine the appropriate character test macros.

You can use either characters or integers as arguments, but the macros are defined only over the integer range from -1 to 255. The functions, however, will correctly handle the entire integer range.

The reason -1 is included as a valid argument is to avoid a nonsense result if you feed the EOF value to one of the macros or functions. EOF can be returned by getchar and other I/O functions, and if you pass it to any of the character test functions, the resulting truth value will be zero.

SEE

ctype

EXAMPLE

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

int main(void)
{
  char b[100];
  int c;

  while((c = getchar()) != EOF)
    printf("\n%c %s alpha.\n",c,
      isalpha(c) ? "is" : "is not");
  return 0;
}
```

isatty

Check if file is a terminal

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

ret = isatty(fh);

int ret;    0 if not a terminal
int fh;     file handle
```

DESCRIPTION

This function returns a non-zero value if the specified file handle is attached to a terminal (TTY) device, i.e. console, printer or auxiliary device.

RETURNS

The return value is 0 if the file is not a terminal or if an error occurred while attempting to obtain the file's characteristics. You can check errno and _OSERR for detailed error information. If the file is a terminal, a value of 1 is returned.

SEE

_disatty, errno, _OSERR

iskbhit, kbhit

Check for keyboard hit

Class: Lattice *Category: Console and Port I/O*

SYNOPSIS

```
#include <dos.h>

hit = iskbhit();
hit = kbhit();

int hit;     0 => no keyboard character ready
             non-zero => character can be read
```

DESCRIPTION

The iskbhit and kbhit functions are part of a group of functions that perform I/O operations with the keyboard and display attached as the console device.

The iskbhit and kbhit functions returns zero if no keyboard character is ready to be read via getch or getche. A non-zero return indicates that a character can be read.

They will also report that a character is waiting if one has been pushed onto the stack with ungetch.

RETURNS

As noted above.

SEE

cgets, cputs, getch, getche, putch, ungetch

labs — Long integer absolute value

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

al = labs(l);

long int l;       long integer
long int al;      absolute value of l
```

DESCRIPTION

The labs function computes the absolute value of long integers, returning a long result.

SEE

abs, fabs, iabs

ldexp

Load exponent

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <math.h>

v = ldexp(f,x);

double v;       value
double f;       fraction
int x;          exponent
```

DESCRIPTION

The ldexp function adds the integer x to the exponent in f, which is the same as computing:

```
v = f * (2 ** x)
```

Note that if f and x are the results of frexp, then ldexp performs the reverse operation. Also, if the absolute value of the resulting exponent is greater than 1023, then matherr will be called with an overflow or underflow error indication.

SEE

fmod, frexp, matherr, modf

_LinkerDB

Pointer to static merged data section

Class: Lattice *Category: Linker Defined Symbols*

SYNOPSIS

```
extern __far _LinkerDB;
```

DESCRIPTION

The address of this external variable is used by the startup code to locate the static copy of the merged data section so that the global base register (A4) may be set. Note that if a program is to be made resident or may have multiple copies running then A4 will not point to the same place as _LinkerDB but to a local copy of the merged data.

localeconv

Numeric formatting convention inquiry

Class: ANSI *Category: Localisation*

SYNOPSIS

```
#include <locale.h>

localeconv();

struct lconv; numeric formatting information
```

DESCRIPTION

The localeconv function sets the components of an object with type struct lconv with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The localeconv function gives a programmer access to information about how to format numeric quantities. The members of the structure, each with type char*, are pointers to strings, any of which (except decimal_point) can point to " ", to indicate that the value is not available in the current locale or is of zero length. The members with type char are non-negative numbers, any of which can be CHAR_MAX to indicate that the value is not available in the current locale. The members include the following:

`char *decimal_point;`	The decimal-point character used to format non-monetary quantities.
`char *thousands_sep;`	The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.
`char *grouping;`	A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.
`char *positive_sign;`	The string used to indicate a nonnegative-valued formatted monetary quantity.
`char *negative_sign;`	The string used to indicate a negative-valued formatted monetary quantity.
`char *mon_grouping;`	A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *int_curr_symbol;`	The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217 Codes for the Representation of Currency and Funds. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.
`char *currency_symbol;`	The local currency symbol used to format monetary quantities.
`char int_frac_digits;`	The number of fractional digits (those after the decimal point) to be displayed in an internationally formatted monetary quantity.
`char frac_digits;`	The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.
`char p_cs_precedes;`	Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.
`char p_sep_by_space;`	Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.
`char n_cs_precedes;`	Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a negative formatted monetary quantity.
`char n_sep_by_space;`	Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char *mon_decimal_point;`	The decimal-point used to format monetary quantities.
`char *mon_thousands_sep;`	The separator for groups of digits before the decimal-point in formatted monetary quantities.
`char n_sep_by_space;`	Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
`char p_sign_posn;`	Set to a value indicating the positioning of the positive_sign for a nonnegative formatted monetary quantity.
`char n_sign_posn;`	Set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity.

The elements of grouping and mon_grouping are interpreted according to the following:

CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of p_sign_posn and n_sign_posn is interpreted according to the following:

Value	Placement of sign string
0	precedes the quantity and currency_symbol.
1	precedes the quantity and currency_symbol.
2	succeeds the quantity and currency_symbol.
3	immediately precedes the currency_symbol.
4	immediately succeeds the currency_symbol.

RETURNS

The localeconv function returns a pointer to the filled-in object. The structure pointed to by the return value must not be modified by the program, but may be overwritten by a subsequent call to the localeconv function. In addition, calls to the setlocale function with categories LC_ALL, LC_MONETARY, or LC_NUMERIC may overwrite the contents of the structure.

EXAMPLE

The following table illustrates the rules which may well be used by four countries to format monetary quantities:

Country	Positive format	Negative format	International format
Italy	L.1.234	-L.1.234	ITL.1.234
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by localeconv are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK"	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_thousands_sep	""	","	","	"."
mon_grouping	""	""	""	""
positive_sign	"-"	"-"	"-"	"C"
negative_sign	""	""	""	""
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

localtime — Unpack Greenwich Mean Time to local time

Class: ANSI

SYNOPSIS

```
#include <time.h>

ut = localtime(t);

struct tm *ut;   unpacked time
const time_t *t; packed time
```

DESCRIPTION

The localtime function unpacks a time value from the time_t form into a structure. Normally the time value represents the number of seconds since 00:00:00, January 1, 1970, Greenwich Mean Time. The time function (described elsewhere) returns this kind of number. Using the localtime function, this number is adjusted for the local time zone.

The localtime function uses the _tzset function to set environmental variables for its time zone conversions.

Note that the localtime function expects a pointer as the argument. A common error is to pass the actual time value instead of the pointer.

Also, localtime and gmtime share a static data area for their return values. A call to either one will destroy the results of the previous call.

SEE

asctime, ctime, gmtime, time, _tzset, utpack, utunpk

log, log10

Logarithmic functions

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

r = log(x);      Natural logarithm functions
r = log10(x);    Base 10 logarithm functions

double r;        result
double x;        argument
```

DESCRIPTION

The log and log10 functions take the base e and base 10 logarithm, respectively. Each of these requires a positive argument. If a negative argument is supplied, matherr will be called with a DOMAIN error.

SEE

exp, matherr, pow, sqrt

Class: Lattice *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

length = lprintf(fmt,arg1,arg2,...);

int length;        number of characters generated
const char *fmt;   format string
```

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 lprintf form of printf sends output to the stdprt file, which is usually a line printer.

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, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf

_lrotl, _lrotr

Rotate long integers

Class: Microsoft *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

left = _lrotl(value,count);
right = _lrotr(value,count);

unsigned long left;       left rotated value
unsigned long right;      right rotated value
unsigned long value;      value for rotation
int count;                rotation count
```

DESCRIPTION

The _lrotl and _lrotr functions rotate the long integer value to the left or right (respectively) by the number of bits specified by the count argument. This differs from the standard shift operators (<< and >>) in that the bits from the top of the longword are not lost, but replace the lower bits and vice-versa.

Note that this function is normally implemented using a #pragma inline.

RETURNS

The value rotated as required.

SEE

_rotl, _rotr

Class: OLD *Category: Memory Block Manipulation*

SYNOPSIS

```
#include <stdlib.h>

p = lsbrk(lbytes);

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

DESCRIPTION

The lsbrk function allocates a large block from the linear heap. This heap is viewed as a contiguous memory region with allocated space at its lower end and free space above that. A "break pointer" contains the address of the first free location. The lsbrk function increments or decrements this break pointer.

RETURNS

For lsbrk, an error is indicated by a NULL pointer.

SEE

getmem, malloc, rbrk, sbrk

lsearch, lfind

Linear search and update

Class: UNIX *Category: Search and Sort*

SYNOPSIS

```
#include <stdlib.h>

match = lsearch(key,base,pnel,size,(*cmp)(obj,arr));
match = lfind(key,base,pnel,size,(*cmp)(obj,arr));

void *match;          matched element or NULL pointer
const void *key;      object to be matched
const void *base;     initial element of searched array
size_t *pnel;         pointer to number of elements
size_t size;          size of each element
int (*cmp)();         comparison function
const void *obj;      pointer to key
const void *arr;      pointer to an array element
```

DESCRIPTION

The lsearch function searches an array of *pnel objects (the initial element of which is pointed to by base) for an element that matches the object pointed to by key. The size of each element of the array is specified by size.

The comparison function pointed to by cmp is called with two arguments that point to the key object and to an array element, in that order. The function returns an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array element.

If the element cannot be found in the table the integer *pnel is incremented and the datum added at the end of the array.

The lfind function searches the array in the same way as lsearch, but the datum is not added if the search fails.

RETURNS

The lsearch function returns a pointer to a matching element of the array. The lfind function will return a NULL pointer if no match is found. If two elements compare as equal, the element matched will be the first in the array.

SEE

bsearch

lseek, tell

Set or get file position

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

apos = lseek(fh,rpos,mode); set unbuffered file
                            position
apos = tell(fh);            get unbuffered file
                            position

int fh;                     file handle
long rpos;                  relative file position
int mode;                   seek mode
long apos;                  absolute file position
```

DESCRIPTION

The lseek function moves the byte cursor of an unbuffered 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.

If lseek is asked to move 0 bytes relative to the current position, it simply returns the current file position. The tell function is then equivalent to:

```
apos = lseek(fh,0L,SEEK_CUR);
```

RETURNS

Both functions return -1L if an error occurs, in which case errno and _OSERR contain additional error information.

SEE

Fseek, errno, _OSERR, open

EXAMPLE

```
/*
 * This program totals the number of bytes used by
 * all normal files in the current directory.
 */

#include <fcntl.h> /* for unbuffered I/O */

char names[8192]; /* holds file names */

int main(void)
{
  char *p;
  int f,n;
  long x,y;

  if(getfnl("*.*",names,sizeof(names),0) <= 0)
  {
    printf("Can't build file name list\n");
    exit(1);
  }
  for(x = 0, n = 0, p = names; *p; p += strlen(p) + 1)
  {
    f = open(p,O_RDONLY);
    if(f < 0)
    {
      printf("Can't open \"%s\"\n",p);
      exit(1);
    }
    y = lseek(f,0L,2);
    if(y < 0)
    {
      printf("Seek failure on \"%s\"\n",p);
      exit(1);
    }
    x += y;
    n++;
    close(f);
  }
  printf("%d files, %ld bytes used\n",n,x);
}
```

Class: ANSI *Category: Process Creation*

SYNOPSIS

```
ret = main(argc,argv,envp);

int ret;            program termination code
int argc;           argument count
char *argv[];       argument vector
char *envp[];       environment vector
```

DESCRIPTION

This function does not actually exist in the library; you must supply one of these "main programs" in each of your applications. If you trace through the two startup modules C.S and _MAIN.C, you will find that C.S passes control to _MAIN.C, which then calls the function named main. Since we supply the source code for both of these modules, you are free to change this initialisation procedure for special applications. The standard version simulates UNIX's interface with C programs by setting up two "vectors", which are simply arrays of pointers.

The argv array contains pointers to the command line arguments, and argc indicates how many pointers are in the array. For example, if you invoke myprog with the following command line:

```
myprog abc def "ghi jkl"
```

then argv is set up as follows:

```
argv[0] => "myprog" with extended command lines
        => "" for standard GEMDOS
argv[1] => "abc"
argv[2] => "def"
argv[3] => "ghi jkl"
```

and argc contains the value 4.

The envp array contains pointers to the environment strings, and the array is terminated with a NULL pointer. Environment strings are normally created via the putenv function, and each one has the following format:

```
name=variable
```

While envp is provided for compatibility with UNIX (and does not exist in ANSI), you should normally use the getenv function to find environment names. This is particularly important if you add strings to the environment via the putenv function, because putenv may re-allocate the enviroment pointer vector, and so the original envp will no longer be correct.

There is an external variable named environ which starts out the same as envp and gets updated whenever putenv moves the vector. In summary, use envp only if you do not use putenv within your program.

RETURNS

When main returns to its caller (normally _MAIN.C), the program exits via the exit function passing the value returned from main to it. Alternatively you may explicitly call the exit function with a termination code.

Heed the above warnings about the use of envp.

SEE

environ, exit, getenv, putenv, _exit

malloc

Allocate a memory block

Class: ANSI *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

b = malloc(n);

void *b;       block pointer
size_t n;      number of bytes
```

DESCRIPTION

The malloc function allocates a block that is n bytes long and is aligned in such a way that you can cast the block pointer to any pointer type. If the block cannot be allocated, a NULL pointer is returned.

RETURNS

The malloc function returns a pointer to the block. A NULL pointer is returned if there is not enough space for the requested block.

If you need space for a string, be sure to use strlen(string)+1 to allow room for the null.

SEE

calloc, realloc, free, getmem, rlsmem, sbrk

Class: UNIX *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

a = matherr(x);          math error handler
r = except(type,name,arg1,arg2,retval);
                         call maths error handler

int a;                   action code
struct exception *x;     exception vector
double r;                actual return value
int type;                error type
char *name;              maths function name
double arg1;             first argument
double arg2;             second argument
double retval;           proposed return value
```

DESCRIPTION

The matherr function is called whenever one of the higher-level maths functions detects an error. The exception vector structure is defined in math.h and contains information about the error as follows:

```
struct exception
{
  int type;              error type
  char *name;            maths function name
  double arg1, arg2;     function arguments
  double retval;         proposed return value
};
```

The standard library version of matherr translates the error type into a UNIX error code that is placed into errno. Then the function returns an action code of 0 to indicate that the maths function should simply use the proposed return value. In other words, the maths function will pass that value back to its caller.

The Lattice compiler package includes the source code to matherr so that you may change it to do more sophisticated error correction if required. One typical change is to place a different return value into the exception vector and then return a non-zero action code. This informs the maths function that the return value has been changed.

The except function is a Lattice extension to UNIX that simplifies the interface to matherr by setting up the exception vector and processing the action code and return value. It is intended to ease the error-handling chore in user-written maths functions.

When your maths function encounters an error, it should call except specifying one of the following error types, which are defined in the math.h header file:

Symbol	Code	Meaning
DOMAIN	1	Domain error
SING	2	Singularity
OVERFLOW	3	Overflow (number too large)
UNDERFLOW	4	Underflow (number too small)
TLOSS	5	Total loss of significance
PLOSS	6	Partial loss of significance

You can define new type codes if your application requires them, but you should then change matherr to perform the appropriate mapping into the UNIX error codes. The default mapping is:

matherr	errno
DOMAIN	EDOM
SING	EDOM
OVERFLOW	ERANGE
UNDERFLOW	ERANGE
TLOSS	ERANGE
PLOSS	ERANGE

RETURNS

For matherr, a non-zero return indicates that the proposed return value in the exception vector has been changed and that the new value should be used. A zero return indicates that the proposed return value is OK.

For except, the actual return value (a double) is passed back.

SEE

_CXFERR

Class: UNIX *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

v = max(a,b); Compute maximum of two values
v = min(a,b); Compute minimum of two values
```

DESCRIPTION

These functions compute the maximum and minimum of two arithmetic values.

Note that two versions of max and min are available, one from math.h implemented as a macro (for any type) and one from string.h (for type int only) as a builtin function. The statement #include <string.h> provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement.

Class: ANSI *Category: Wide Characters*

SYNOPSIS

```
#include <stdlib.h>

num = mblen(s,n);

int num;         number of bytes
const char *s;   array of multibyte characters
size_t n;        bytes of array to check
```

DESCRIPTION

If s is not a NULL pointer, the mblen function determines the number of bytes comprising the multibyte character pointed to by s. Except that the shift state of the mbtowc function is not affected, it is equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

RETURNS

If s is a NULL pointer, the mblen function returns a zero value, if multibyte character encodings do not have state-dependent encodings, otherwise non-zero to indicate that the encodings are state-dependent. If s is not a NULL pointer, then mblen either returns 0 (if s points to the null character), or returns the number of bytes that comprise the multibyte character (if the next n or fewer bytes form a valid multibyte character), or -1 (if they do not form a valid multibyte character).

SEE

mbtowc

mbstowcs — Convert sequence of multibyte characters

Class: ANSI *Category: Wide Characters*

SYNOPSIS

```
#include <stdlib.h>

num = mbstowcs(pwcs,s,n);

size_t num;       number of array elements modified
wchar_t *pwcs;    array to contain codes
const char *s;    array containing multibyte characters
size_t n;         number of characters to convert
```

DESCRIPTION

The mbstowcs function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by s into a sequence of corresponding codes and stores not more than n codes into the array pointed to by pwcs. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the mbtowc function, except that the shift state of the mbtowc function is not affected.

No more than n elements will be modified in the array pointed to by pwcs.

RETURNS

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

mbtowc — Determine number of bytes of multibyte character

Class: ANSI *Category: Wide Characters*

SYNOPSIS

```
#include <stdlib.h>

num = mbtowc(pwc,s,n);

int num;            number of bytes
wchar_t *pwc;       object to store codes
const char *s;      array containing multibyte characters
size_t n;           number of characters to check
```

DESCRIPTION

If s is not a NULL pointer, the mbtowc function determines the number of bytes that comprise the multibyte character pointed to by s. It then determines the code for the value of type wchar_t that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and pwc is not a NULL pointer, the mbtowc function stores the code in the object pointed to by pwc. At most n bytes of the array pointed to by s will be examined.

RETURNS

If s is a NULL pointer, the mbtowc function returns a non-zero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a NULL pointer, the mbtowc function either returns 0 (if s points to the null character), or returns the number of bytes that comprise the converted multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than n or the value of the MB_CUR_MAX macro.

Class: ANSI

SYNOPSIS

```
#include <string.h>

s = memccpy(to,from,c,n);   Copy a memory block up to
                            a character
s = memchr(a,c,n);          Find a character in a
                            memory block
x = memcmp(a,b,n);          Compare two memory blocks
s = memmove(to,from,n);     Move a memory block
s = memcpy(to,from,n);      Copy a memory block
s = memset(to,c,n);         Set a memory block to a
                            value
s = memswp(a,b,n);          Swap two memory blocks
s = memrep(a,b,n,n);        Replicate values through a
                            block

movmem(from,to,m);          Move a memory block
repmem(to,vt,nv,nt);        Replicate values through a
                            block
setmem(to,m,c);             Set a memory block to a
                            value
swmem(a,b,m);               Swap two memory blocks

void *to;                   destination pointer
const void *from;           source pointer
unsigned m;                 number of bytes
size_t n;                   number of bytes
int c;                      character value
void *a,*b;                 block pointers
char *vt;                   value template
int nv;                     number of bytes in
                            template
int nt;                     number of templates in
                            block
void *s;                    return pointer
int x;                      return value
```

DESCRIPTION

These functions manipulate blocks of memory in various ways.

The memmove and movmem functions are similar, except the former was introduced with UNIX V, while the latter is a traditional Lattice function. In a like manner, memset and setmem perform the same operation, except that the former is UNIX-compatible. Note that memcpy and memset return a pointer to the destination block, while movmem and setmem have void returns. Also note that memmove is smart enough to handle overlapping memory blocks correctly.

The memccpy function is similar to memcpy except that copying stops after the specified block size has been copied or after the specified character has been copied. It returns a pointer to the character after c in the from block, or a NULL pointer if c was not found in the first n characters. Note that, like memcpy, memccpy does not handle overlapping memory blocks. If you specify overlapping blocks to this function, the results are unpredictable.

The memchr function returns a pointer to the first occurrence of the specified character in the block, or a NULL pointer if the character is not found.

The memcmp function performs a character-by-character comparison of two memory blocks and returns an integral value as follows:

Return	Meaning
Negative	First block is 'less-than' second
Zero	First block equals second
Positive	First block is 'greater-than' second

There is no UNIX equivalent for swmem and repmem. The former merely swaps two blocks in memory, although it has a major performance advantage over the typical for-loop approach. The latter replicates a template of values throughout a block and is very useful when you need to initialise an array of structures to some non-zero pattern. The memswp and memrep are provided to give a more ANSI like interface to the swmem and repmem functions.

Note that memcmp, memcpy, and memset have built-in versions which are functionally equivalent to the standard library versions. A built-in version generates in-line 68000 instructions without needing to make calls to the library. The statement #include <string.h> provides a default setting by which any built-in functions are accessed. If you don't want a particular built-in function, you can use an #undef statement as follows: #undef memcmp.

Note that these functions neither recognise nor produce the null terminator byte usually found at the end of strings. A popular mistake is to assume that memcpy, unlike strcpy, automatically places a null byte at the end of the block. It does not.

When choosing a string function the ANSI mem... functions are preferred over the older Lattice functions which are provided only for backward compatability.

Unlike previous versions of the Lattice C Compiler, memcpy *is not smart enough* to handle overlapping blocks. The ANSI function memmove should be used instead.

RETURNS

As noted above.

Class: UNIX *Category: File System Manipulation*

SYNOPSIS

```
#include <stdio.h>

error = mkdir(path);

int error;              0 if successful
const char *path;       points to new directory path
                        string
```

DESCRIPTION

This function makes a new directory in the specified path. For example, if path is "c:\\abc\\def\\ghi", then the new directory is named "ghi" and is in the path "c:\\abc\\def". The path may begin with a drive letter and a colon.

RETURNS

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

SEE

Dcreate, errno, _OSERR

mktemp

Create a unique filename

Class: UNIX *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

p = mktemp(template);

char *p;            address of template or NULL
char *template;     template string
```

DESCRIPTION

This function creates a unique file name from the template string and returns a pointer to the name. The template string should be a filename in the directory required, terminated by six trailing Xs. mktemp replaces the string "XXXXXX" with a unique code generated from the process id and a unique string.

RETURNS

If the operation is successful, the function returns a pointer to the string. If a unique filename cannot be generated or if the template does not match the specification.

SEE

getpid, tmpfile, tmpnam

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

cal = mktime(timeptr);

time_t cal;              calendar time value
struct tm *timeptr;      time value to be converted
```

DESCRIPTION

The mktime function converts the broken-down time, expressed as local time, in the structure pointed to by timeptr into a calendar time value with the same encoding as that of the values returned by the time function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of tm_mday is not set until tm_mon and tm_year are determined.

RETURNS

The mktime function returns the specified calendar time encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value ((time_t)-1).

EXAMPLE

```
This simple example is a program to determine what
day of the week is
July 11, 2001.

#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
  "Sunday", "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday", "Sunday",
  "-unknown-"
};

struct tm time_str;

  time_str.tm_year = 2001 - 1900;
  time_str.tm_mon = 7 - 1;
  time_str.tm_mday = 11;
  time_str.tm_hour = 0;
  time_str.tm_min = 0;
  time_str.tm_sec = 1;
  time_str.tm_isdst = -1;
  if (mktime(&time_str) == -1)
    time_str.tm_wday = 7;
  printf("%s\n", wday[time_str.tm_wday]);
```

Class: ANSI *Category: Numeric Transformation*

SYNOPSIS

```
#include <math.h>

x = modf(y,p);

double x;    signed fractional part of y
double y;    floating point value.
double *p;   pointer to integral part of y
```

DESCRIPTION

The modf function separates the integral and fractional parts of y and returns them as two doubles. The function return value is the fractional part, and the integral part is placed in the double pointed to by p. Both parts have the same sign as y. Note that the fractional part is the number that would be obtained by calling the fmod function in the following way:

```
x = fmod(y,1.0);
```

Make sure that the second argument of modf is a pointer to a double. A common error is to use a pointer to an integer.

SEE

Refer to fmod for an example involving modf.

_MSTEP

Memory pool increment size

Class: Lattice *Category: Memory Management*

SYNOPSIS

```
extern unsigned long _MSTEP;
```

DESCRIPTION

This external integer is used by the memory allocation functions. It specifies the minimum amount of memory that will be allocated from the system when additional memory is required for the local memory pool.

When additional memory is added to the local pool, it will not be contiguous with the memory already in the pool. If the additional amount is small, it can lead to severe fragmentation of the local pool. The memory allocation functions attempt to avoid this by rounding the amount needed up to the next multiple of the figure in _MSTEP.

Note that when the value in this variable is zero the startup code sizes it in such a way as to avoid any GEMDOS memory allocation problems, hence in general you should not adjust the value.

onbreak

Plant break trap

Class: Lattice *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
#include <dos.h>

error = onbreak(func);

int error;              error return
int (*func)(void);      function to register
```

DESCRIPTION

This function plants a break trap, which is a user-supplied function that gets called whenever the user keys Ctrl-C, whenever any console I/O is being performed. The function can use any operating system services, since it is not really called as an interrupt routine. Note that under this implementation the program is always aborted after processing of the function registered via onbreak.

If func is NULL, then the current break trap, if any, is removed and the default interrupt handler is restored. With the default handler, Ctrl-C causes a program abort.

RETURNS

The onbreak function returns 0 if it was successful. The break trap function should return non-zero to abort for compatability with other systems, although in this implementation the abort always occurs.

EXAMPLE

```
/*
 * This program tests the onbreak function. After the
 * initial message is printed, you should get the
 * "Break received" message if you hit Ctrl-C.
 * If you hit any other character, the program will
 * exit, printing "Successful"
 */

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

int brk(void)     /* This is the break function */
{
  printf("Break received...\n");
  return 1;
}
```

```
int main(void )  /* This is the main program */
{
  printf("Setting break trap...\n");
  if(onbreak(brk))
    printf("Can't set break trap\n");
  for (;;)
    if(kbhit())
      break;
  printf("Successful\n");
}
```

Class: Lattice *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
#include <stdlib.h>

success = onexit(func);

int success;          non-zero if successful
int (*func)(int);     pointer to trap function
```

DESCRIPTION

This function establishes a "trap" that will be called when the program terminates. The trap function is called just before the program returns to the operating system. For normal termination via the exit function or via a return from the main function, all buffers are flushed and files are closed before the trap is called. If the program is using _exit, the files and buffers may still be open, depending on what the program does before terminating. In both cases, user-allocated memory is not yet freed.

This function is similar to the ANSI function atexit, however the exit code is passed as a parameter to the trap function as its only argument. Then whatever value the trap function returns is used as the real exit code. Also only one such trap may exist. Each call to onexit overrides the previous trap. If you call onexit with a NULL pointer, the current trap is removed.

Remember that the exit trap is called after all files have been closed, unless the program is terminating via _exit. This means that the keyboard and screen devices normally associated with file handles 0, 1, and 2 will no longer be accessible. A common mistake is to issue some type of output message via printf or cprintf from within the exit trap. In order for this to work, you should fopen or open the con: device and send the message via fprintf or write.

SEE

atexit, exit, _exit

EXAMPLE

```
/*
 * This program tests the "onexit" function.
 */

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

int ex(int i) /* This is the exit trap function */
{
  FILE *con;

  if((con = fopen("con:","w")) != NULL)
    fprintf(con,"Exit trap hit...code %d found\n",i);
  return 0;
}

int main(void) /* This tests the exit trap */
{
  int (*p)(int);

  p = ex;
  printf("Setting exit trap...\n");
  if(!onexit(p))
    printf("Can't set trap...\n");
  printf("Exiting with code 2\n");
  exit(2);
}
```

open

Open an unbuffered file

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

fh = open(name,mode,prot);

int fh;              file handle
const char *name;    file name
int mode;            access mode
int prot;            protection mode (O_CREAT only)
```

DESCRIPTION

This function opens a file so that it can be accessed via the unbuffered I/O functions. The name can be any valid file name, and it may include a device code and a directory path. The access mode is formed by ORing together the appropriate symbols from the following list:

<table>
<tr><td>O_RDONLY</td><td>Read-only access. No writes are allowed.</td></tr>
<tr><td>O_WRONLY</td><td>Write-only access. No reads are allowed.</td></tr>
<tr><td>O_RDWR</td><td>Read-write access. Both reads and writes are allowed.</td></tr>
<tr><td>O_CREAT</td><td>If the file does not already exist, it is created with the protection mode specified by prot. The protection mode specified via the symbols S_IREAD and S_IWRITE, which are defined in fcntl.h:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>S_IWRITE</td><td>Write allowed</td></tr>
<tr><td>S_IREAD</td><td>Read allowed</td></tr>
<tr><td>S_IWRITE | S_IREAD</td><td>Both allowed</td></tr>
<tr><td>0</td><td>Both allowed</td></tr>
</table>
If the file already exists the prot argument is ignored. Also, you can use chgfa or chmod to change the protection bits after the file has been closed.</td></tr>
<tr><td>O_APPEND</td><td>This symbol is normally used in conjunction with O_WRONLY or O_RDWR. It causes the I/O system to seek to the end of the file before each write operation. After each write operation, the file is positioned at the new end-of-file.</td></tr>
</table>

O_TRUNC	If the file exists, it is truncated to a length of 0. This flag is normally used with O_CREAT, O_WRONLY or O_RDWR.
O_NDELAY	This symbol is defined for UNIX compatibility and has no effect under GEMDOS.
O_EXCL	This symbol is used only with O_CREAT. If O_EXCL and O_CREAT are both present and the file already exists, the open function will fail.
O_RAW	The file is read and/or written with no translation. Without this flag, the external integer named _iomode is consulted, and if it contains zero, the file is translated. This means that carriage returns ('\r') are dropped on input and are inserted before line feeds ('\n') on output.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

Fopen, Fcreate, errno, _OSERR, chgfa, chmod, close, creat

opendir, closedir

Open/Close a directory stream

Class: POSIX *Category: Directory Manipulation*

SYNOPSIS

```
#include <dirent.h>

dir = opendir(name);
closedir(dir);

DIR *dir;               directory handle
const char *name;       file name
```

DESCRIPTION

The opendir family of functions allow system independent processing of directories. The opendir function opens the directory specified by name and returns a pointer to an associated directory stream, dir, or NULL if the directory cannot be opened.

The closedir function closes the stream dir and frees any resources which were allocated by the opendir function.

RETURNS

The opendir function returns a pointer to an associated directory descriptor, or the value NULL if the directory was not found or enough memory could not be allocated to hold the directory structure or buffer.

SEE

readdir, rewinddir, seekdir, telldir, getfnl, dfind, dnext

Class: Lattice *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

fh = opene(name,mode,prot,path);

int fh;                  file handle
const char *name;        file name
int mode;                unbuffered file access mode
int prot;                protection mode
char *path;              path return
```

DESCRIPTION

The opene function is like open 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 or handle. 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 or handle. If unsuccessful, continue.

- Find the environment variable named PATH and repeat the preceding step with those directory names. If unsuccessful, return an error indication.

See the description of the fopene function for an example of opene.

RETURNS

If the operation is successful, the function returns a file handle, which is an integer equal to or greater than 0. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

fopen, fopene, open

Class: GEMDOS *Category: Errors*

SYNOPSIS

```
#include <dos.h>

extern long volatile _OSERR;   GEMDOS error code
extern int os_nerr;            number of error codes
extern char *os_errlist[];     GEMDOS error messages
```

DESCRIPTION

The external integer named _OSERR contains error information returned by GEMDOS after a system call has failed. In general, the Lattice library resets _OSERR at the beginning of any function that makes GEMDOS system calls. Then if a system call fails during that function, the system error code is saved in _OSERR.

The GEMDOS error number is mapped into an equivalent UNIX error number, which is placed in errno. If there is no appropriate UNIX number, errno will contain -1, defined symbolically as EOSERR. The function returns with a suitable error indication, which is usually -1 for functions that return integer values or NULL for functions that return pointers.

The os_nerr and os_errlist items are defined in a C source file named oserr.c and are used by the poserr function to print messages that correspond to the code found in _OSERR.

The following list applies to all current versions of GEMDOS and is what is provided in oserr.c:

Symbol	Code	Meaning
ERROR	01	"Fundamental error"
EDRVNR	02	"Drive not ready"
EUNCMD	03	"Unknown command"
E_CRC	04	"Data error"
EBADRQ	05	"Bad request structure length"
E_SEEK	06	"Seek error"
EMEDIA	07	"Unknown media type"
ESECNF	08	"Sector not found"
EPAPER	09	"Printer paper alarm"

EWRITF	10	"Write fault"
EREADF	11	"Read fault"
EWRPRO	13	"Can't write on protected device"
E_CHNG	14	"Invalid disk change"
EUNDEV	15	"Unknown unit"
EBADSF	16	"Bad sectors on format"
EOTHER	17	"Insert other disk"
EINVFN	32	"Invalid function number"
EFILNF	33	"File not found"
EPTHNF	34	"Path not found"
ENHNDL	35	"Too many files opened"
EACCDN	36	"Access denied"
EIHNDL	37	"Invalid handle"
ENSMEM	39	"Insufficient memory"
EIMBA	40	"Invalid memory block address"
EDRIVE	46	"Invalid drive code"
ENSAME	48	"Not same device"
ENMFIL	49	"No more files"
E_RANGE	64	"Range error"
EINTRN	65	"GEMDOS internal error"
EPLFMT	66	"Invalid program load format"
EGSBF	67	"Memory growth failure"

SEE

poserr

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
#include <basepage.h>

BASEPAGE *_pbase;
```

DESCRIPTION

This external pointer points to the basepage of the current process. In general you should not manipulate the elements of this directly, but instead allow the operating system to do it for you.

The structure pointed to has the following *public* elements:

```
typedef struct _base
{
   void           *p_lowtpa;      bottom of TPA
   void           *p_hitpa;       top of TPA + 1
   void           *p_tbase;       base of text segment
   long           p_tlen;         length of text
   void           *p_dbase;       base of data segment
   long           p_dlen;         length of data
   void           *p_bbase;       base of BSS segment
   long           p_blen;         length of BSS
   void           *p_dta;         current DTA pointer
   struct _base   *p_parent;      parent's basepage
   void           *p_reserved;
   char           *p_env;         environment strings
   long           p_undef[20];
   char           p_cmdlin[128];  command line image
} BASEPAGE;
```

Note that although further information is available within this structure it is *not* public and if you attempt to access it your program may not work with future versions of the OS.

SEE

Pexec

perror

Class: ANSI *Category: Errors*

SYNOPSIS

```
#include <stdio.h>

perror(s);

const char *s;    message prefix
```

DESCRIPTION

This function checks errno and, if it is non-zero, prints an error message on stderr. The message consists of the specified prefix, a colon and space, and the message text from the external array named sys_errlist. This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of external integer sys_nerr. The Lattice compiler package contains the source for these two external items in a file named syserr so you can change or expand the messages as you desire. See the description of errno for a list of the current error messages.

SEE

errno, sys_nerr, sys_errlist, poserr

popen, pclose

Open a pipe to/from a process

Class: UNIX *Category: Process Creation*

SYNOPSIS

```
#include <stdio.h>

fp = popen(cmd,mode);
err = pclose(fp);

int err;                  error return value
FILE *fp;                 file pointer
const char *cmd;          command to execute
const char *mode;         file access mode
```

DESCRIPTION

The popen and pclose functions initiate a pipe to the named command, or close the pipe respectively. The argument cmd is a command passed to system to which the data is to be sent, or received from. The mode specifies whether the command is to be used as an input or output filter. If mode is "r" then the data is collected from the processes standard output, otherwise if the mode is "w" the data written to fp is sent to the processes standard input.

The pclose function cleans up the buffers used by the popen function and returns the exit status of the command called.

Note that under UNIX this command causes concurrent execution of the called process and it's parent, whereas under GEMDOS the called command is always a executed as the single active process.

RETURNS

The function popen returns a file handle fp associated with the stream if the command could be successfully completed otherwise the value NULL.

The pclose function returns 0 if the process was successfully closed, otherwise the value -1 is returned and an appropriate value placed in errno. Note that pclose may fail if it cannot find the required command and the stream was opened for write mode.

SEE

errno, system

perror

Print UNIX error message

Class: ANSI *Category: Errors*

SYNOPSIS

```
#include <stdio.h>

perror(s);

const char *s;   message prefix
```

DESCRIPTION

This function checks errno and, if it is non-zero, prints an error message on stderr. The message consists of the specified prefix, a colon and space, and the message text from the external array named sys_errlist. This array contains pointers to the various UNIX error messages. The highest error number is given by the contents of external integer sys_nerr. The Lattice compiler package contains the source for these two external items in a file named syserr so you can change or expand the messages as you desire. See the description of errno for a list of the current error messages.

SEE

errno, sys_nerr, sys_errlist, poserr

popen, pclose

Open a pipe to/from a process

Class: UNIX *Category: Process Creation*

SYNOPSIS

```
#include <stdio.h>

fp = popen(cmd,mode);
err = pclose(fp);

int err;                 error return value
FILE *fp;                file pointer
const char *cmd;         command to execute
const char *mode;        file access mode
```

DESCRIPTION

The popen and pclose functions initiate a pipe to the named command, or close the pipe respectively. The argument cmd is a command passed to system to which the data is to be sent, or received from. The mode specifies whether the command is to be used as an input or output filter. If mode is "r" then the data is collected from the processes standard output, otherwise if the mode is "w" the data written to fp is sent to the processes standard input.

The pclose function cleans up the buffers used by the popen function and returns the exit status of the command called.

Note that under UNIX this command causes concurrent execution of the called process and it's parent, whereas under GEMDOS the called command is always a executed as the single active process.

RETURNS

The function popen returns a file handle fp associated with the stream if the command could be successfully completed otherwise the value NULL.

The pclose function returns 0 if the process was successfully closed, otherwise the value -1 is returned and an appropriate value placed in errno. Note that pclose may fail if it cannot find the required command and the stream was opened for write mode.

SEE

errno, system

EXAMPLE

```
/*
 * collect the output from the dir command
 * will fail if 'dir' cannot be found
 */

#include <stdio.h>

void showdir(void)
{
  FILE *fp;
  char buf[100];

  fp=popen("dir","r");
  if (fp)
  {
    while (fgets(buf,sizeof(buf),fp))
      printf("%s, ",buf);
    pclose(fp);
  }
}
```

Class: GEMDOS *Category: Errors*

SYNOPSIS

```
#include <dos.h>

error = poserr(s);

int error;          contents of _OSERR
const char *s;      message prefix
```

DESCRIPTION

This function checks _OSERR and, if it is non-zero, sends an error message to stderr. The message consists of the specified prefix, a colon and space, and the message text from the external array named os_errlist. This array contains pointers to the various error messages. The highest error number is given by the contents of external integer os_nerr. The Lattice compiler package contains the source for these two external items in a file named oserr.c so you can change or expand the messages as you desire.

RETURNS

The function returns the contents of _OSERR so you can test for an error condition and print a message in one step.

SEE

_OSERR, os_errlist, os_nerr, perror

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

length = printf(fmt,arg1,arg2,...);

const char *fmt;      format string
```

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 printf form sends the output stream to the buffered file named stdout, which is usually the user's screen (i.e., the "console").

The fmt argument points to a string consisting of ordinary characters and conversion specifications. The ordinary characters are simply copied to the output, but each conversion specification is replaced by the results of the conversion. These results come from operating sequentially upon the arguments that follow fmt. That is, the first conversion specification operates upon arg1, the second operates upon arg2, and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent sign (%). If you want to place a percent sign into the output stream, precede it with another percent sign in the fmt string. That is, %% will send a single percent sign to the output stream.

If a percent sign is not followed by another percent, then it introduces a conversion specification, as follows:

```
%[flags][width][.precision][size]type
```

where the brackets [...] indicate optional fields, and the fields have the following definitions:

flags	Controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.
width	Specifies the "field width", which is the minimum number of characters to be generated for this format item.

precision	Specifies the "field precision", which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the type field.
size	Can be either 'l' for "large size" or 'h' for small size. The h comes from UNIX implementations where it means "half-word".
type	Specifies the type of argument conversion to be done.

If any flag characters are used, they must appear immediately after the percent and can be any of the following:

Minus (-)	This causes the result to be left-adjusted within the field specified by width or within the default width.
Plus (+)	This flag is used in conjunction with the various numeric conversion types to cause a plus or minus sign to be placed before the result. If it is absent, the sign character is generated only for a negative number.
Blank	This flag is similar to the plus, but it causes a leading blank for a positive number and a minus sign for a negative number. If both the plus and the blank flags are present, the plus takes precedence.
Hash (#)	This flag causes special formatting. With the 'o', 'x', and 'X' types, the sharp flag prefixes any non-zero output with 0, 0x, or 0X, respectively. The 'p' and 'P' types are treated like 'x' and 'X', respectively. That is, their output is preceded by 0x or 0X if the special formatting flag is present. With the 'f', 'e', and 'E' types, the hash flag forces the result to contain a decimal point. With the 'g' and 'G' types, the hash flag forces the result to contain a decimal point and also prevents the elimination of trailing zeroes.

The width is a non-negative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless width begins with a zero. In that case, zero-padding is performed. Note that width specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the precision specifier for that purpose.

If you don't want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique.

The meaning of the precision item depends on the field type, as follows:

Type c, n, p, P	The precision item is ignored.
Types d, o, u, x, and X	The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.
Types e, E, and f	The precision is the number of digits to appear after the decimal point. If fewer digits are generated, trailing zeroes are supplied.
Types g and G	The precision is the maximum number of significant digits.
Type s	The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

The conversion type can be any of the following:

c	The associated argument must be an integer. The single character in the rightmost byte of the integer is copied to the output.
d	The associated argument must be an integer, and the result is a string of digit characters preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer. If the "large size" modifier is present, the argument is taken as a long integer.
e	The associated argument must be a double, and the result has the form: `-d.ddde-ddd` where d is a single decimal digit, dd is one or more digits, and ddd is an exponent of exactly three digits. The first minus sign is omitted if the floating point number is positive, and the second minus sign is omitted if the exponent is positive. The plus and blank flags dictate whether there will be a sign character emitted if the number is positive. The "large size" modifier is ignored.
E	This is exactly the same as type e except that the result has the form: `-d.dddE-ddd`
f	The associated argument must be a double, and the result has the form `-dd.dd` where dd indicates one or more decimal digits. The minus sign is omitted if the number is positive, but a sign character will still be generated if the plus or blank flag is present. The number of digits before the decimal point depends on the magnitude of the number, and the number after the decimal point depends on the requested precision. If no precision is specified, the default is six decimal places. If the precision is specified as 0, or if there are no non-zero digits to the right of the decimal point, then the decimal point is omitted.
g	The associated argument must be a double, and the result is in the 'e' or 'f' format, depending on which gives the most compact result. The 'e' format is used only when the exponent is less than -4 or greater than the specified or default precision. Trailing zeroes are eliminated, and the decimal point appears only if any non-zero digits follow it.

G	This is identical to the 'g' format, except that the 'E' type is used instead of 'e'.
I	The associated argument is taken as a signed integer. The corresponding argument will be a pointer to an integer. If the "large size" modifier is present, the argument must be a long integer.
n	The associated argument is taken to be a pointer to an integer. The integer reflects the number of characters written to the output to this point in the printf call. No argument is converted.
o	The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits. If the "large size" modifier is present, the argument must be a long integer.
p	The associated argument is taken as a data pointer, and it is converted to hexadecimal representation.
P	This is the same as the 'p' format, except that upper case letters are used as hexadecimal digits.
s	The associated argument must point to a null-terminated character string. The string is copied to the output, but the null byte is not copied.
u	The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits. If the "large size" modifier is present, the argument must be a long integer.
x	The associated argument is taken as an unsigned integer, and it is converted to a string of hexadecimal digits with lower case letters. If the "large size" modifier is present, the argument is taken as a long integer.
X	This is the same as the 'x' format, except that upper case letters are used as hexadecimal digits.

RETURNS

This function returns the number of output characters generated.

SEE

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

EXAMPLE

```
/*
 * This example prints a message indicating whether
 * the function argument is positive or negative.
 * In the second "printf", the width and precision
 * are 15 and 8, respectively.
 */

#include <stdio.h>

void pneg(double value)
{
  char *sign;

  if(value < 0)
    sign = "negative";
  else
    sign = "not negative";

  printf("The number %E is %s.\n",value,sign);

  printf("The number %*.*E is %s.\n",15,8,value,sign);
}
```

putc, putchar

Put a character to a buffered file/stdout

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

r = putc(c,fp);
r = putchar(c);

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

DESCRIPTION

The putc function puts a single character to the specified file previously opened via fopen, freopen, or fdopen. Whereas putchar writes the character to the standard output file. Note that they are actually implemented as macros in order to maximise execution speed.

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, fputc, fputchar, freopen, _OSERR

putch

Put char to console

Class: Lattice *Category: Console and Port I/O*

SYNOPSIS

```
#include <dos.h>

a = putch(c);

int a;      character written to the console or EOF
int c;      character to write
```

DESCRIPTION

The putch function is one of a group of functions that perform I/O operations with the keyboard and display attached as the console device.

The putch function simply writes the specified character to the display screen at the current cursor position. When accessed in this way, the screen behaves like a "glass TTY". That is, the carriage return, line feed, and backspace characters behave as they would on a simple printer. Alas, the form feed character does not clear the screen.

RETURNS

The function returns the character written to the console if successful, or EOF if the character could not be written.

SEE

cgets, cputs, getch, getche, kbhit, ungetch

putenv

Put string into environment

Class: UNIX *Category: Process Environment*

SYNOPSIS

```
#include <stdlib.h>

error = putenv(env);

int error;     0 if successful
char *env;     environment string
```

DESCRIPTION

The putenv function accepts a string that has the form

```
name=var
```

and places it into the current environment. If the environment already contains a string beginning with name= then that string is replaced; otherwise, the new string is added.

After putenv is called, the original envp argument that was passed to your main program may no longer be valid. However, the external data item named environ does get updated when necessary, and is therefore valid at all times. Also note that the string env is added to the environment, and should not be subsequently used as a parameter to free.

RETURNS

A non-zero return value from putenv indicates that the environment could not be expanded in size to accept the new string.

SEE

environ, getenv, rmenv

EXAMPLE

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

if(putenv("HOCUS=pocus")) /* Add HOCUS */
  fprintf(stderr,"Couldn't add HOCUS\n");

putenv("HOCUS=");       /* Remove HOCUS */

rmvenv("HOCUS");        /* Another way to remove it */
```

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

error = puts(s);

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

DESCRIPTION

The puts function copies string s to stdout, the standard output file. The terminating null byte is not copied, but a newline is sent after the string.

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, fputs

EXAMPLE

The following example writes two lines to the standard output file, stdout. It demonstrates how the fputs function, which takes a file pointer argument, can be used to mimic the puts function.

```
#include <stdio.h>

puts("This is the first line");
fputs("This is ",stdout);
puts("the second line");
```

qsort, et al

Sort a data array

Class: ANSI *Category: Search and Sort*

SYNOPSIS

```
#include <stdlib.h>

qsort(a,n,size,cmp);   Sort a data array
dqsort(da,n);          Sort an array of doubles
fqsort(fa,n);          Sort an array of floats
lqsort(la,n);          Sort an array of long integers
sqsort(sa,n);          Sort an array of short integers
tqsort(ta,n);          Sort an array of text pointers

void *a;               data array pointer
double *da;            pointer to double array
float *fa;             pointer to float array
long *la;              pointer to long int array
short *sa;             pointer to short int array
char *ta[];            pointer to text pointer array

size_t n;              number of elements in array
size_t size;           element size in bytes
int (*cmp)(const void *,const void *);
                       pointer to comparison function
```

DESCRIPTION

The qsort function sorts the specified data array using the quicksort algorithm. During its operation, it calls upon the specified comparison routine with pointers to the two array elements being compared. The comparison routine should return an integral result as follows:

Return	Meaning
Negative	First element is below second
Positive	First element is above second
Zero	Elements are equal

The dqsort, fqsort, lqsort, sqsort and tqsort functions sort various arrays which are commonly encountered. They are all straightforward except for tqsort, which requires some explanation. The ta array consists of pointers to null-terminated character strings. The tqsort function re-arranges the pointers so that the strings are in ascending ASCII sequence, using strcmp as the comparison routine. Note that the sort is based on the contents of the strings rather than their physical address.

EXAMPLE

```
/*
 * sort an array of strings using qsort
 */

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

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

void sort(char *s[],size_t n)
{
  qsort(s,n,sizeof(*s),cmp);
}
```

Class: ANSI *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
#include <signal.h>

err=raise(sig);

int err;          error status
int sig;          signal to assert
```

DESCRIPTION

The raise function sends the signal sig to the executing program. This is functionally identical to calling a user-supplied routine that is related to the signal number.

RETURNS

The raise function returns 0 if successful, non-zero if unsuccessful.

SEE

signal

Class: ANSI *Category: Random Numbers*

SYNOPSIS

```
#include <stdlib.h>

x = rand();
srand(seed);

unsigned int seed;    random number seed
int x;                random number
```

DESCRIPTION

The rand function returns pseudo-random numbers in the range from 0 to the maximum positive integer value. The random number generator can be reset to a new seed value by calling the srand function. The initial default seed is 1.

See drand48 and its related functions for more sophisticated random number generation.

RETURNS

As noted above.

SEE

drand48, srand

EXAMPLE

```
/* This example prints 1000 random numbers.*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

  if(argc > 1)
  {
    stcd_i(argv[1],&x);
    printf("Seed value is %d\n",x);
    srand(x);
  }
  printf("Here are 1000 random numbers...\n");
  for(i = 0; i < 200; i++)
    printf("%5d %5d %5d %5d %5d\n",
      rand(),rand(),rand(),rand(),rand());
}
```

read, write

Read or write a unbuffered file

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <fcntl.h>

cnt = read(fh,buf,length);
                         Read from unbuffered file
cnt = write(fh,cbuf,length);
                         Write to unbuffered file

size_t cnt;          actual bytes read or written
int fh;              file handle
const void *cbuf;    data buffer
void *buf;           data buffer
size_t length;       number of bytes to read or write
```

DESCRIPTION

These functions read or write an unbuffered file whose handle was returned by creat or open. Under normal circumstances, the value returned should match the buffer length. If this value is -1 or greater than the requested length, then some type of error occurred, and you should consult errno and _OSERR. If the actual length is less than the requested length when reading, this usually means that the file is exhausted. Similarly, if the actual length is less than the requested length for a write operation, this usually means that the device has no more space available. In both of these cases, it is still a good idea to check errno and _OSERR just in case some malfunction caused the short count.

Note that these functions are very similar to the functions _dread and _dwrite. The differences are that unbuffered files will be automatically closed by exit and _exit, which are usually called for you when the program terminates, and that all translation occurs at this level.

RETURNS

If the operation is successful, the function returns the actual number of bytes transferred. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

errno, _OSERR, open, _dread, _dwrite

Class: POSIX *Category: Directory Manipulation*

SYNOPSIS

```
#include <dirent.h>

ent = readdir(dir);

struct dirent *ent; pointer to directory entry
DIR *dir;           directory handle
```

DESCRIPTION

The readdir function returns a pointer to the next directory entry, or NULL on reaching the end of the directory structure.

The pointer returned, ent, is only guaranteed to contain the element d_name, giving the name of the file. Under GEMDOS this structure contains further information and is defined as:

```
struct dirent
{
    int d_attr;              /* GEMDOS file attribute */
    time_t d_time;           /* time */
    size_t d_size;           /* file size */
    char d_name[FMSIZE];     /* directory entry name */
};
```

RETURNS

The readdir function returns a pointer to the next directory entry, or the value NULL if all entries have been read.

SEE

closedir, opendir, rewinddir, seekdir, telldir, getfnl, dfind, dnext

EXAMPLE

```
/*
 * search for a file in a directory
 */

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

```
int find_file(const char *s,const char *where)
{
  DIR *dir;
  struct dirent *dp;

  dir=opendir(where);
  while (dp=readdir(dir))
    if (!strcmp(dp->d_name,s))
    {
      closedir(dir);
      return 1;   /* file found */
    }
  closedir(dir);
  return 0; /* file not found */
}
```

realloc

Re-allocate a memory block

Class: ANSI *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

nb = realloc(b,n);

void *b;        block pointer
size_t n;       number of bytes
void *nb;       new block pointer
```

DESCRIPTION

This function reallocates a block, changing its size. The original block is copied to the new one. If the new block is smaller, then the upper part of the original block is not copied.

RETURNS

If successful, realloc returns a pointer to the new block. A NULL pointer is returned if there is not enough space for the requested block.

SEE

calloc, malloc, free

remove, unlink

Remove a file

Class: ANSI

SYNOPSIS

```
#include <stdio.h>

error = remove(name);        remove a file
error = unlink(name);        remove a file

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

DESCRIPTION

These functions remove the specified file from the system. They behave identically, but unlink is provided for compatibility with some versions of UNIX. The remove function is preferred because it is now in the ANSI C standard.

The name argument can include a path, but it cannot include wild card characters. That is, you can remove only one file at a time.

RETURNS

If a non-zero value is returned, some type of error occurred, and additional information can be found in errno and _OSERR. The most common errors occur when you try to remove a file that doesn't exist or that is marked as read-only.

SEE

errno, _OSERR

EXAMPLE

```
/*
 * This program removes all files specified in the
 * argument list. It does not allow wild card
 * characters in the file names.
 */

#include <stdio.h>

int main(int argc,char *argv[])
{
  int i; /* loop counter */
  int ret = 0;    /* exit code */

  for(i = 1; i < argc; i++)
    if(remove(argv[i]))
    {
      perror("RMV");
      ret = 1;
    }
  return ret;
}
```

rename

Rename a file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

error = rename(old,new);

int error;          0 for success, -1 for error
const char *old;    old file name
const char *new;    new file name
```

DESCRIPTION

This function renames a file, if possible. If the new file name includes a directory path that is different than that of the old name, the file is disconnected from the old directory and connected to the new one. For example, after executing this statement:

```
rename("\\olddir\\file","\\newdir\\file");
```

you will no longer find file in the olddir directory.

RETURNS

If the function fails, it returns -1 and places additional error information into errno and _OSERR. Success is indicated by a return value of 0.

SEE

Frename

EXAMPLE

```
/*
 *
 *  This is a version of the RENAME command
 *  that prompts for the old and new names.
 *
 */
#include  <stdlib.h>
#include  <stdio.h>
#include  <dos.h>

int main(int argc,char *argv[])
{
  char  old[FMSIZE],new[FMSIZE];
  char  *pold,*pnew;

  if(argc < 2) /* Get old file name */
  {
    printf("OLD FILE:  ");
    if(gets(old) == NULL)
      exit(1);
    pold = old;
  }
  else
    pold = argv[1];

  if(arc < 3)
  {
    printf("NEW FILE:  ");
    if(gets(new) == NULL)
      return 1;
    pnew = new;
  }
  else
    pnew = argv[2];

  if(rename(pold,pnew))
  {
    perror("RENAME");
    return 1;
  }
  return 0;
}
```

rewind

Seek to beginning of buffered file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

rewind(fp);

FILE *fp;    file pointer
```

DESCRIPTION

The rewind macro is implemented as an fseek call. The rewind macro resets the specified file to its first byte and is equivalent to the following fseek call:

```
fseek(fp,0L,0);
```

where the second argument indicates relative position (0) and the third argument represents mode (0 for relative to the beginning of the file).

See the description of fseek for information on its use and return values.

SEE

errno, fopen, fseek, ftell, lseek, _OSERR, tell

rlsmem, rlsml

Release a memory block

Class: OLD *Category: Memory Block Manipulation*

SYNOPSIS

```
#include <stdlib.h>

error = rlsmem(p,sbytes);
error = rlsml(p,lbytes);

int error;                    non-zero if error
void *p;                      block pointer
unsigned sbytes;              number of bytes
size_t lbytes;                number of bytes
```

DESCRIPTION

These functions release memory blocks that were previously obtained via getmem or getml.

RETURNS

If the block is not in the current memory pool or overlaps a block that is already free, a value of -1 is returned. Otherwise, the return value is 0.

Class: UNIX *Category: File System Manipulation*

SYNOPSIS

```
#include <stdio.h>

error = rmdir(path);

int error;          0 if successful
const char *path;   points to directory path string
```

DESCRIPTION

This function removes an existing directory in the specified path. For example, if path is "c:\\abc\def\\ghi", then the directory named "ghi" is removed from the path "c:\\abc\\def". The path may begin with a drive letter and a colon.

RETURNS

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

SEE

Ddelete, errno, _OSERR

rmvenv

Remove environment string

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
#include <stdlib.h>

error = rmvenv(envname);

int error;              0 if successful
const char *envname;    environment name string
```

DESCRIPTION

The rmvenv function accepts a string that specifies the name of an environment variable. If that name exists, then it is removed from the environment. The envname argument can also be a constructed as:

```
name=var
```

and the function will simply ignore everything after the equal sign. See putenv for an example involving rmvenv.

RETURNS

For rmvenv, a non-zero return indicates that the specified name is not currently defined in the environment.

SEE

environ, getenv, putenv

_rotl, _rotr

Rotate short integers

Class: Microsoft *Category: Numeric Transformation*

SYNOPSIS

```
#include <stdlib.h>

left = _rotl(value,count);
right = _rotr(value,count);

unsigned short left;      left rotated value
unsigned short right;     right rotated value
unsigned short value;     value for rotation
int count;                rotation count
```

DESCRIPTION

The _rotl and _rotr functions rotate the short integer value to the left or right (respectively) by the number of bits specified by the count argument. This differs from the standard shift operators (<< and >>) in that the bits from the top of the word are not lost, but replace the lower bits and vice-versa.

Note that this function is normally implemented using a #pragma inline.

RETURNS

The value rotated as required.

SEE

_lrotl, _lrotr

Class: OLD *Category: Memory Management*

SYNOPSIS

```
#include <stdlib.h>

p = sbrk(sbytes);

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

DESCRIPTION

The sbrk function allocates a short block from the linear heap. This heap is viewed as a contiguous memory region with allocated space at its lower end and free space above that. A "break pointer" contains the address of the first free location. The sbrk function increments or decrements this break pointer.

RETURNS

If sbrk fails, it returns value -1 cast to a generic pointer (void *). This strange return is a legacy of UNIX.

SEE

getmem, lsbrk, malloc, rbrk

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

n = scanf(fmt,arg1,arg2,...);

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

DESCRIPTION

The scanf function performs formatted input conversions on text obtained from the standard input file. The input characters are read and checked against the format string, which may contain any of the following:

White space

Any number of spaces, horizontal tabs, or newline characters will cause input to be read up to the next character that is not white space.

Ordinary characters

Any character that is not white space and is not the percent sign (%) must match the next input character. Use a double percent (%%) in the format string to match a single percent in the input. If there is not an exact match, scanning stops, and the function returns.

Conversion specification

This is is multi-character sequence that indicates how the next input characters are to be converted. The form is:

```
%*nlt
```

where the various fields are defined as follows:

%	A percent sign introduces a conversion specifier. If you want to match a percent sign in the input, indicate this by a double percent (%%) in the format string.
*	The asterisk is optional. If present, it means that the conversion should be performed, but the result should not be stored. There should be no value pointer in the argument list for a suppressed conversion.
n	This is an optional decimal number that specifies the maximum input field width. This is used only with the s format.
h	The letter 'h' is optional. If present, it indicates that a short conversion should be performed.
l	The letter 'l' is optional. If present, it indicates that a long conversion should be performed.
t	The t stands for one of the following format characters: c, d, e, f, g, i, n, o, s, u, x. These are described below.

If the conversion is successful and assignment is not suppressed, the result is placed into the corresponding argument. The argument list must contain a pointer to an appropriate data item for each conversion specification that does not suppress assignment.

The function returns the number of conversion values that were assigned. This can be less than the number expected if the input characters do not agree with the format string. If an end-of-input is reached before *any* values are assigned, the return value is EOF.

The format characters listed above specify how the input characters are to be converted. Leading white space is skipped in all cases except the (, c, and n conversions.

c	The corresponding argument must point to a character. The next input character is moved to that destination. Note that *no* white space is skipped.
d	The corresponding argument must point to an integer or to a long integer. The latter applies if the d is preceded by an l. The input characters must be decimal digits, optionally preceded by a plus or minus sign.

e,f,g	These three types are identical. The corresponding argument must point to a float or a double. The latter applies if the type letter is preceded by an 'l'. The input characters must consist of the following sequence: Optional leading white space. An optional plus (+) or minus (-). A sequence of decimal digits. An optional decimal point followed by 0 or more decimal digits. An optional exponent, consisting of the letter 'e' or 'E' followed by an optional plus or minus sign followed by 1 or more decimal digits. This general form is shown below, where [...] indicates an optional part: `[space][sign]digits[.digits][exponent]`
i	A signed integer is expected. The corresponding argument must point to a signed integer or a signed long integer if the 'i' is preceded by an 'l'. This specifier is similar to 'd' but it will additionally interpret numbers specified in other than decimal format.
n	No input characters are read. The corresponding argument must point to an integer into which is written the number of input characters read so far.
o	An octal number is expected, and the corresponding argument should point to an integer, or to a long integer if the 'o' is preceded by an 'l'.
p	The associated argument is taken as a data pointer, and it is converted from a hexadecimal representation.
s	A character string is expected, and the corresponding argument should point to a character array large enough to hold the string and a terminating null byte. The input string is terminated by white space or the end-of-input. Also, if a maximum field width is specified, the output array size should be at least that width plus 1, because the reading of input characters will stop at the field width even if no white space has been hit.

u	An unsigned decimal number is expected, and the corresponding argument should point to an unsigned integer, or to an unsigned long integer if the 'u' is preceded by an 'l'.
x	A hexadecimal number is expected, and the corresponding argument should point to an integer, or to a long integer if the 'x' is preceded by an 'l'. The hexadecimal number can begin with the characters "0x" or "0X", and case is not significant for the hexadecimal letters.
(	A nonempty sequence of characters from the given "scanset" is expected. The corresponding argument should point to the initial character of a character array large enough to hold the sequence and a terminating null byte. The conversion specifier includes all subsequent characters ("scanlist") in the format string, up to and including the right bracket. Also consider the following special cases with the caret symbol (^): If the ^ character is used as the first one after the left bracket, the scanset contains all character that do not appear between the brackets. If the conversion specifier () or (^) is used, the right bracket itself is in the scanlist and the next right bracket character is the matching right one that ends the specification; otherwise the first right bracket is the one that ends the specification. If a - character in the scanlist is not first, second after the ^ character, or last in order, the scanlist contains the range from the characters before and after the - character, inclusive.

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.

All of the result arguments (i.e. arg1, arg2, and so on) must be pointers. Also, you should not supply a pointer for any conversion specification that uses the * to suppress assignment.

SEE

cscanf, fscanf, sscanf

seekdir, rewinddir, telldir Seek on directory entries

Class: POSIX *Category: Directory Manipulation*

SYNOPSIS

```
#include <dirent.h>

seekdir(dir,pos);     seek to new directory position
pos = telldir(dir);   find current directory position
rewinddir(dir);       move to start of directory

DIR *dir;             directory handle
long pos;             directory position
```

DESCRIPTION

The seekdir function sets the position where the next readdir operation will occur. The position should be one previously obtained from the telldir function which returns the current position.

The rewinddir macro, simply resets the directory position to the start of the directory.

RETURNS

The telldir function returns a long value giving the current position of the associated directory stream.

SEE

closedir, opendir, readdir, getfnl, dfind, dnext

_setargv

Parse command line arguments

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
__regargs char **_setargv(char *line, char **argv);

char *line;         null terminated command line
char **argv;        argument vector to fill in
```

DESCRIPTION

The _setargv is called during the startup code to parse the command line arguments. You may replace this if you wish with your own code if you wish to say perform wild card matching of arguments. Note that this function *must* be declared as a register passing function and compiled without stack checks.

Also note that this function will never be called if the command was passed a pre-parsed command line using the Atari extended command line format.

The parameter line is a null terminated command line which the routine should parse, storing pointers to the parsed arguments at argv upwards. The final value of argv after parsing is then returned. The source code to the standard _setargv module is supplied in the package.

RETURNS

The value you return from this function is a pointer to the first free byte above the area into which you parsed the arguments.

setbuf

Set file buffer

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

setbuf(fp,buff);

FILE *fp;      file pointer
char *buff;    buffer pointer
```

DESCRIPTION

The setbuf function sets the buffering mode for a file previously opened via fopen, freopen, or fdopen. You should call the function immediately after opening the file. If you fail to follow this rule, the file may become corrupted.

The buffered I/O system automatically allocates a buffer via malloc when you perform the first read or write operation. Then the data being read or written is staged through this buffer in order to improve I/O efficiency. If you would rather use your own buffer instead of having one allocated for you, call setbuf with a non-NULL buffer pointer. The buffer size must be at least as large as the value given in the external integer _bufsiz, which defaults to the value of the symbol BUFSIZ, defined in stdio.h.

You can eliminate buffering and still use the buffered I/O functions by calling setnbf or by calling setbuf with a NULL buffer pointer. When this is done, physical I/O occurs whenever your program performs buffered read or write operation, even if only one byte is being transferred. This is very inefficient for disk files, but often desirable for terminal or communication ports.

The setbuf function must be used only after fopen, freopen, or fdopen and before any other buffered file operations. Also, a common error is to allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack.

SEE

fopen, freopen, fdopen, setnbf, setvbuf

setjmp, longjmp

Set long jump parameters

Class: ANSI *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
#include <setjmp.h>

ret = setjmp(save);
longjmp(save,value);

int ret;          return code
int value;        return value
jmp_buf save;     save area
```

DESCRIPTION

The setjmp function checkpoints the current stack mark in the save area and returns a code of 0. A subsequent call to longjmp will then cause control to return to the next statement after the original setjmp call, with value as the return code. If value is 0, it is forced to 1 by longjmp.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances. Structured programming gurus lose a lot of sleep over the "pathological connections" that can result from indiscriminate usage of these functions.

RETURNS

A return code of 0 from setjmp indicates that this is the initial call to save the stack.

Calling longjmp with an invalid save area is an effective way to disrupt the system. One common error is to use longjmp after the function calling setjmp has returned to its caller. This cannot possibly succeed, since the stack frame for that function no longer exists.

Note that since the Lattice C compiler performs automatic register allocation the only automatic variables guaranteed to remain valid are those explicitly declared volatile. Consider the function:

```
#include <setjmp.h>

jmp_buf j;

int f(void)
{
  int x;

  x=f1();
  if (setjmp(j))
    return x;
```

```
        x=f2();

        return  f3(x);
    }
```

If in this function a longjmp occurs in f3 the value of x may or may not be restored to the value at the setjmp. If this is important the variable x should be declared:

```
    volatile  int  x;
```

so that the value of x after a longjmp will be that which was in force from the assignment from f2.

setlocale

Set locale control parameters

Class: ANSI *Category: Localisation*

SYNOPSIS

```
#include <locale.h>

old = setlocale (category,locale);

char *old;               pointer to old locale
int category;            category to change
const char *locale;      new environment
```

DESCRIPTION

The setlocale function provides the mechanism for controlling locale-specific features of the library. The category argument allows parts of the library to be localised as necessary without changing the entire locale-specific environment. Specifying the locale argument as a string gives an maximum flexibility in providing a set of locales. For instance, an implementation could map the argument string into the name of a file containing appropriate localisation parameters; these files could then be added and modified without requiring any recompilation of a localisable program.

The setlocale function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The setlocale function may be used to change or query the program's entire current locale or portions thereof. The value LC_ALL for category names the program's entire locale; the other values for category name only a portion of the program's locale. LC_COLLATE affects the behaviour of the strcoll and strxfrm functions. LC_CTYPE affects the behaviour of the character-handling functions and the multibyte functions. LC_MONETARY affects the monetary formatting information returned by the localeconv function. LC_NUMERIC affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the non-monetary formatting information returned by the localeconv function. LC_TIME affects the behaviour of the strftime function.

A value of "C" for locale specifies the minimal environment for C translation: a value of " " for locale specifies the native environment.

At program startup, the equivalent of:

```
set locale(LC_ALL, "C");
```

is executed.

RETURNS

If a pointer to a string is given for locale and the selection can be honoured, the setlocale function returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honoured, the setlocale function returns a NULL pointer and the program's locale is not changed.

A NULL pointer for locale causes the setlocale function to return a pointer to the string associated with the category for the program's current locale; the program's locale is not changed.

The pointer to string returned by the setlocale function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to cannot be modified by the program, but may be overwritten by a subsequent call to the setlocale function.

SEE

localeconv, strcoll, strftime, strxfrm

setnbf

Set non-buffer mode for a file

Class: UNIX *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

error = setnbf(fp);

int error;    0 upon success
FILE *fp;     file pointer
```

DESCRIPTION

The setnbf function sets the unbuffered mode for a file previously opened via fopen, freopen, or fdopen. You should call the function immediately after opening the file. If you fail to follow this rule, the file may become corrupted.

By calling this function, the buffering is eliminated, but you may still use the buffered I/O functions. When this is done, physical I/O occurs whenever your program performs buffered read or write operation, even if only one byte is being transferred. This is very inefficient for disk files but often desirable for terminal or communication ports.

The setnbf functions must be used only after fopen, freopen, or fdopen and before any other buffered file operations.

SEE

fopen, freopen, fdopen, setbuf, setvbuf

setvbuf

Set variable file buffer

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

error = setvbuf(fp,buff,type,size);

int error;          0 if successful
FILE *fp;           file pointer
char *buff;         buffer pointer
int type;           type of buffering
size_t size;        buffer size in bytes
```

DESCRIPTION

The setvbuf function sets the buffering mode for a file previously opened via fopen, freopen, or fdopen. You should call the function immediately after opening the file. If you fail to follow this rule, the file may become corrupted.

The setvbuf function can do everything that the other two functions (setbuf and setnbf) can do, and it can also set "line buffered" mode and attach a buffer of non-standard size. The type argument must be one of the following symbols defined in stdio.h:

Value	Meaning
_IOFBF	Fully buffered
_IOLBF	Line buffered
_IONBF	Non-buffered

For _IOFBF and _IOLBF, the specified buffer will be attached to the file unless buff is NULL, in which case a buffer will be automatically allocated on the first read or write. For the _IONBF case, the buff and size arguments are ignored.

The line-buffered mode is useful for interactive applications. When in this mode, the buffer is flushed whenever a newline is sent, the buffer is full, or input is requested. Note, however, that you must use the fputc and fputchar functions instead of the putc and putchar macros in order for line buffering to work correctly. The macros do not check if line-buffered mode is active, and so they behave as if the file were fully buffered.

The setvbuf function must be used only after fopen, freopen, or fdopen and before any other buffered file operations. Also, a common error is to allocate a buffer on the stack within a function, attach it to a file, and then return from the function. This will corrupt the stack.

RETURNS

For setvbuf, the error code is non-zero if type or size is invalid.

SEE

fopen, freopen, fdopen, setbuf, setnbf

signal

Establish event traps

Class: ANSI *Category: Non-Local Jumps/Signal Handling*

SYNOPSIS

```
#include <signal.h>

oldfun = signal(sig,newfun);

int (*oldfun)();      old trap function
int sig;              signal number
int (*newfun)();      new trap function
```

DESCRIPTION

This function establish traps for various events that can occur outside of your program. The newfun argument specifies the action to be taken when the signal occurs, as follows:

SIG_IGN	Ignore the signal.
SIG_DFL	Take the system default action for each signal.

If newfun is not any of the above, then it must be a valid function pointer. When the signal is detected, the action is reset to either SIG_DFL or SIG_IGN, depending on the particular signal. Then the trap function is called with an integer argument specifying which signal was detected (e.g. SIGINT). The trap function can take whatever action is necessary, including calling signal again to re-establish itself as the trap function. If the function returns, execution continues at the point in your program where the signal was detected.

The sig argument specifies which signal is being trapped, using the symbols defined in signal.h.

RETURNS

The signal function normally returns the previous value of the trap function, which may be SIG_IGN or SIG_DFL. It may return SIG_ERR to indicate an attempt to set an illegal signal number.

SEE

raise

_SLASH

Directory separator character

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
extern char _SLASH;
```

DESCRIPTION

This external character is used by various functions which construct file names. It specifies the character to be used for separating components of the directory path. For GEMDOS and MSDOS it is a backslash (\), whilst under UNIX and AmigaDOS it is a slash (/).

SEE

strmfn, strmfp

sizmem

Get memory pool size

Class: OLD *Category: Memory Block Manipulation*

SYNOPSIS

```
#include <stdlib.h>

size = sizmem();

long size;
```

DESCRIPTION

This function returns the number of unallocated bytes in the current memory pool. This value is the sum of the sizes of all unallocated blocks, and so it does not indicate the size of the largest free block.

Also, the value does not indicate the maximum amount of memory that can be allocated. That is, the allocation functions will automatically expand the pool when no block of sufficient size is found in the pool.

SEE

getmem, getml, rlsmem, rlsml, rstmem

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

length = sprintf(s,fmt,arg1,arg2,...);

int length;       number of characters generated
const char *fmt;  format string
char *s;          storage 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 sprintf form of printf places the output characters into the storage area whose address is given by s. You must ensure that this area is large enough to hold the maximum number of characters that might be generated. Note that sprintf also generates a null byte to terminate the stored string.

See the description of the printf function for a complete discussion of the arguments and conversion specifications. An example is also provided.

RETURNS

This function returns the number of output characters generated. For sprintf, this number does not include the terminating null byte.

SEE

cprintf, fprintf, lprintf, printf, vfprintf, vprintf, vsprintf

sscanf

Formatted input from a string

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

```
#include <stdio.h>

n = sscanf(ss,fmt,arg1,arg2,...);

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

DESCRIPTION

The sscanf function performs formatted input conversions on text obtained from a string. 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.

SEE

cscanf, fscanf, scanf

_STACK, _STKDELTA — Stack specification

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
extern unsigned long _STACK;        stack size
extern unsigned long _STKDELTA;     'chicken' factor
```

DESCRIPTION

This external value _STACK is used by the startup code to define the initial stack space allocated to the process. To increase it from the default 4k, you should include an initialised variable of the form:

```
unsigned long _STACK=16384;
```

in your program. The associated variable _STKDELTA sets the minimum 'distance' which the stack checking code will allow between the top of the data area and the bottom of the stack before calling _xcovf.

SEE

_base, _xcovf

stat

Get status of named file

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

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

ret = stat(name,statbuf);

int ret;                 0 if successful
const char *name;        path naming a file
struct stat *statbuf;    stores information about file
```

DESCRIPTION

The stat function returns UNIX-style file status information about the file specified by name. The buffer returned is defined in sys/stat.h as follows:

```
struct stat
{
  dev_t st_dev;                 disk drive number
  ino_t st_ino;                 inode number (not used)
  unsigned short st_mode;       file mode flags
  short st_nlink;               number of links (always 1)
  short st_uid;                 user id (not used)
  short st_gid;                 group id (not used)
  dev_t st_rdev;                same as st_dev
  off_t st_size;                file size in bytes
  time_t st_atime;              time of last access
  time_t st_mtime;              time of last modification
  time_t st_ctime;              time of creation
};
```

Note that the header file sys/types.h must be included prior to sys/stat.h as this defines the types dev_t, ino_t, dev_t and off_t.

RETURNS

On success, the stat function returns 0.

stcarg

Get an argument

Class: Lattice *Category: Argument Processing*

SYNOPSIS

```
#include <string.h>

length = stcarg(s,b);

size_t length;      number of bytes in argument
const char *s;      text string pointer
const char *b;      break string pointer
```

DESCRIPTION

This function scans the text string until one of the break characters is found or until the null terminating byte is hit. While scanning, stcarg skips over substrings that are enclosed in single or double quotes, and the backslash is recognised as an escape character. In other words, break characters will not be detected if they are quoted or preceded by a backslash.

RETURNS

The function returns a count of the number of characters in s up to but not including the break character or null terminator.

SEE

stpbrk,strcspn,strpbrk

EXAMPLE

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

int main(void)
{
  char a[256],b[256];
  int x;

  for (;;)
  {
    printf("Enter text string: ");
    if(gets(a) == NULL)
      return 0;
    printf("Enter break string: ");
    if(gets(b) == NULL)
      return 0;
    x = stcarg(a,b);
    printf("Length: %d, Text: \"%.*s\"\n\n",x,x,a);
  }
}
```

Class: Lattice *Category: Numeric Transformation*

SYNOPSIS

```
#include <string.h>

length = stcd_i(in,ivalue);  decimal string to int
length = stco_i(in,ivalue);  octal string to int
length = stch_i(in,ivalue);  hexadecimal string to
                             int
length = stcd_l(in,lvalue);  decimal string to long
                             int
length = stco_l(in,lvalue);  octal string to long
                             int
length = stch_l(in,lvalue);  hexadecimal string to
                             long

int length;                  input length
const char *in;              input string pointer
int *ivalue;                 integer value pointer
long *lvalue;                long integer value
                             pointer
```

DESCRIPTION

These functions scan an input string and convert the leading characters into short or long integers. For stcd_i and stcd_l, the input string must begin with a plus sign '+', minus sign '-', or a decimal digit ('0' to '9'). The octal conversions stco_i and stco_l process an unsigned string of octal digits ('0' to '7'). Finally, the hexadecimal conversions stch_i and stch_l handle unsigned strings containing digits from '0' to '9' and letters from 'A' to 'F' or 'a' to 'f'. Scanning of the input string stops when the first invalid character is reached, At that point, the resulting value is stored into the area addressed by the second argument.

RETURNS

Each function returns the number of input characters converted. This result will be 0 if the first character of the input string is not valid for the particular conversion. In that case, conversion result stored via the second argument will be 0.

EXAMPLE

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

```
int  main(void)
{
  int  x;
  long  j;
  char  b[80];

  for  (;;)
  {
    printf("\nEnter  a  hexadecimal  value:  ");
    if(gets(b)  ==  NULL)
      break;
    x  =  stch_l(b,&j);
    printf("stch_l:  Length  %d,  Result  %lx\n",x,j);
  }
  return  0;
}
```

stcgfe, stcgfn, stcgfp

Get file name components

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

size = stcgfe(ext,name);  Get file extension
size = stcgfn(node,name); Get file node
size = stcgfp(path,name); Get file path

int size;                 size of result string
char *ext;                extension area pointer
char *node;               node area pointer
char *path;               path area pointer
const char *name;         file name pointer
```

DESCRIPTION

These functions isolate the path, node, or extension portion of a file name. The node is the rightmost portion of the file name that is separated from the rest of the name by a colon, slash, or backslash. The extension is the final part of the node that begins with a period, and the path is the leading part of the name up to the node. For example,

Name	Path	Node	Extension
"myprog.c"	""	"myprog.c"	"c"
"\abc.dir\def"	"\abc.dir\"	"def"	""
"\abc.dir\def.ghi"	"\abc.dir\"	"def.ghi"	"ghi"
"c:yourfile"	"c:"	"yourfile"	""
"\abc\"	"\abc\"	""	""

RETURNS

The size value is the same as would be returned by the strlen function. That is, if size is 0, then the desired portion of the file name could not be found and the result area contains a null string.

SEE

strsfn

EXAMPLE

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

int main(void)
{
  char  file[FMSIZE],path[FMSIZE];
  char  node[FMSIZE],ext[FMSIZE];

  while(gets(file) != NULL)
  {
    stcgfe(ext,file);
    stcgfn(node,file);
    stcgfp(path,file);
    printf("PATH: %s NODE: %s EXT: %s",
      path,node,ext);
  }
  return 0;
}
```

Class: Lattice *Category: Numeric Transformation*

SYNOPSIS

```
#include  <string.h>

length  =  stci_d(out,ivalue);    int to decimal
length  =  stci_o(out,ivalue);    int to octal
length  =  stci_h(out,ivalue);    int to hexadecimal
length  =  stcl_d(out,lvalue);    long int to decimal
length  =  stcl_o(out,lvalue);    long int to octal
length  =  stcl_h(out,lvalue);    long int to
                                  hexadecimal
length  =  stcu_d(out,uivalue);   unsigned int to
                                  decimal;
length  =  stcul_d(out,ulvalue);  unsigned long to
                                  decimal

int  length;                      output length
char  *out;                       output buffer pointer
int  ivalue;                      integer value
long  lvalue;                     long integer value
unsigned  int  uivalue;           unsigned integer
                                  value
unsigned  long  ulvalue;          unsigned long integer
                                  value
```

DESCRIPTION

These functions convert various integral values into ASCII strings. The output area must be large enough to accomodate the maximum possible string, including the terminating null byte that each function appends. The following table shows the required lengths.

Function	Length
stci_d	7
stci_o	7
stci_h	5
stcl_d	13

Function	Length
stcl_o	12
stcl_h	9
stcu_d	6
stcul_d	12

For stci_d and stcl_d, the first output character will be a minus sign if the input value is negative. No special leading character is generated if the value is positive. For all functions, leading zeroes are suppressed, and a single '0' character is generated if the input value is 0.

RETURNS

The return value is the number of characters actually placed into the output area, not including the final null byte.

EXAMPLE

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

int main(void)
{
  int i,x;
  char b[13];

  for (;;)
  {
    printf("\nEnter a short integer: ");
    scanf("%d",&i);
    x = stci_d(b,i);
    printf("stci_d: Length %d, Result %s\n",x,b);
    x = stci_o(b,i);
    printf("stci_o: Length %d, Result %s\n",x,b);
    x = stci_h(b,i);
    printf("stci_h: Length %d, Result %s\n",x,b);
  }
}
```

Class: Lattice — *Category: String Search*

SYNOPSIS

```
#include <string.h>

size = stcpm(string,pattern,match);
                                   Unanchored pattern match
size = stcpma(string,pattern); Anchored patter match

size_t size;             size of matching string
const char *string;      string to be scanned
const char *pattern;     pattern string
char **match;            returns pointer to matching
                         string
```

DESCRIPTION

These functions scan a string to find a specified pattern. The pattern is specified in a simplified form of regular expression notation as shown below:

Pattern	Meaning
?	Match any single character
c*	Match 0 or more instances of character c
c+	Match 1 or more instances of character c
\?	Match a question mark (?)
*	Match an asterisk (*)
\+	Match a plus sign (+)

Any other character must match exactly. For example,

Pattern	Matching
"abc"	Only "abc"
"ab*c"	"ac" or "abc" or "abbc" and so on
"ab+c"	"abc" or "abbc" or "abbbc" and so on
"ab?*c"	Any string starting with "ab" and ending with "c"
"ab*c"	Only "ab*c"

Notice that the last pattern requires a double backslash in front of the asterisk. This causes the compiler to place a single backslash in the string so that stcpm or stcpma will see the string as "ab*c".

For stcpma, the match must occur at the beginning of the string, while for stcpm, the match can occur anywhere in the string. In either case, the function returns the size of the matching string or zero if there was no match. Also, stcpm returns a pointer to the beginning of the matching string.

EXAMPLE

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

int main(void)
{
  char s[100],p[100],*r;
  int x;

  for (;;)
  {
    printf("\nSearch string => ");
    if(gets(s) == NULL)
      break;
    printf("Pattern => ");
    if(gets(p) == NULL)
      break;
    x = stcpma(s,p);
    if(x)
      printf("stcpma: %d, \"%.*s\"\n",x,x,s);
    else
      printf("stcpma: no match\n");
    x = stcpm(s,p,&r);
    if(x)
      printf("stcpm: %d, \"%.*s\"\n",x,x,r);
    else
      printf("stcpm: no match\n");
  }
  return 0
}
```

Class: Lattice *Category: String Search*

SYNOPSIS

```
#include <string.h>

q = stpblk(p);

char *q;              updated string pointer
const char *p;        string pointer
```

DESCRIPTION

This function advances the string pointer past white space characters, that is, past all the characters for which isspace is true.

RETURNS

The function returns a pointer to the next non-white-space character. Note that the null terminator byte is not considered to be white space, and so the function will not go past the end of the string.

SEE

stcis, strspn

EXAMPLE

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

int main(void)
{
  char input[256];

  for (;;)
  {
    puts("\nEnter a string with leading blanks...");
    if(gets(input) == NULL)
      break;
    printf("%s\n",stpblk(input));
  }
  return 0
}
```

Class: Lattice — *Category: Date and Time*

SYNOPSIS

```
#include <string.h>

np = stpdate(p,mode,date);

char *np;               updated output string pointer
char *p;                output string pointer
int mode;               conversion mode
const char *date;       date array, as follows
                        date[0] => year - 1980
                        date[1] => month (1 to 12)
                        date[2] => day (1 to 31)
```

DESCRIPTION

This function converts a 3-byte date array into ASCII or BCD according to the `mode` argument:

Mode	Date Format
0	yymmdd (BCD, 3 bytes)
1	yymmdd (ASCII, 7 bytes)
2	mm/dd/yy (ASCII, 9 bytes)
3	mm-dd-yy (ASCII, 9 bytes)
4	MMM d, yyyy (ASCII, up to 13 bytes)
5	Mm...m d, yyyy (ASCII, up to 19 bytes)
6	dd MMM yy (ASCII, 10 bytes)
7	dd MMM yyyy (ASCII, 12 bytes)

In the above formats, `MMM` represents a 3-character month abbreviation in capitals, and `Mm...m` represents the full month name (e.g. January). The `mm`, `dd`, and `yy` terms are 2-character month, day, and year, respectively, while `d` is the date with the leading zero suppressed. The `yyyy` term is the 4-character year obtained by adding 1980 to the first byte of the date array.

For all modes except 0, a null byte is appended to the output string.

RETURNS

The function does not make validity checks on the date array, and so it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE

stptime, getclk, getft, ftunpk

Class: Lattice *Category: String Search*

SYNOPSIS

```
#include <string.h>

p = stpsym(s,sym,symlen);

char *p;            points to next input character
const char *s;      input string
char *sym;          output string
size_t symlen;      sizeof(sym)
```

DESCRIPTION

This function extracts the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string.

The output string is the null-terminated symbol, and will be an empty string if no symbol is found. If the symbol is longer than symlen-1, its excess characters are dropped.

RETURNS

The function returns a pointer to the next character past the symbol.

SEE

stcarg, stpbrk, strcspn, strpbrk

EXAMPLE

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

int main(void)
{
  char a[256],b[10];
  char *p;

  for (;;)
  {
    printf("\nEnter text string: ");
    if(gets(a) == NULL)
      break;
    for (;;)
    {
      p = stpsym(a,b,sizeof(b));
      printf("Symbol: \"%s\" Residual: \"%s\"\n",b,p);
      if(b[0] == '\0')
        break;
    }
  }
  return 0;
}
```

Class: Lattice *Category: Date and Time*

SYNOPSIS

```
#include <string.h>

np = stptime(p,mode,time);

char *np;                updated output string pointer
char *p;                 output string pointer
int mode;                conversion mode
const char *time;        time array, as follows
                         time[0] => hour (0 to 23)
                         time[1] => minute (0 to 59)
                         time[2] => second (0 to 59)
                         time[3] => hundredths (0 to 99)
```

DESCRIPTION

This function converts a 4-byte time array into ASCII or BCD according to the mode argument:

Mode	Time Format
0	hhmmssdd (BCD, 4 bytes)
1	hhmmss (ASCII, 7 bytes)
2	hh:mm:ss (ASCII, 9 bytes)
3	hhmmssdd (ASCII, 9 bytes)
4	hh:mm:ss.dd (ASCII, 12 bytes)
5	hh:mm (ASCII, 6 bytes)
6	hr:mm:ss HH (ASCII, 12 bytes)
7	hr:mm HH (ASCII, 9 bytes)

The hh, mm, ss, and dd terms are simply the 2-digit (BCD or ASCII) equivalents of the binary values in the time array. The hr term is the 2-digit hour using the 12-hour form, and the HH term is either AM or PM.

Note that a null terminator is appended to the ASCII output strings.

RETURNS

The function does not make validity checks on the time array, and so it cannot fail. It returns a pointer to the first byte past the generated output. For modes other than 0, this is a pointer to the null terminator.

SEE

stpdate, getclk, getft, ftunpk

stptok

Get next token from a string

Class: Lattice *Category: String Search*

SYNOPSIS

```
#include <string.h>

p = stptok(s,tok,toklen,brk);

char *p;                points to next character after
                        token
const char *s;          points to input string
char *tok;              points to output buffer
size_t toklen;          sizeof(tok)
const char *brk;        break string
```

DESCRIPTION

This function breaks out the next token from the input string and moves it to the token buffer with a null terminator. A token consists of all characters in the input string s up to but not including the first character that is in the break string. In other words, brk specifies the characters that cannot be included in a token.

If the input string begins with a break character, then the token buffer will contain a null string, and the return pointer p will be the same as s. If no break character is found after toklen-1 input characters have been moved to the token buffer, or if the input string terminator (a null byte) is hit, then the scan stops as if a break character were hit.

RETURNS

The function returns a pointer to the next character in the input string.

Note that the function does not delete white space at the beginning of the input string.

SEE

stpblk, strtok

strbpl

Build string pointer list

Class: Lattice *Category: String Search*

SYNOPSIS

```
#include <string.h>

n = strbpl(s,max,t);

long n;             number of pointers
char *s[];          pointer to string pointer list
size_t max;         maximum number of pointers
const char *t;      text pointer
```

DESCRIPTION

This function constructs a list of pointers to the strings contained within the specified text array. Each string must be null-terminated, and the text array must be terminated by a null string. In other words, array t must end with two null bytes, one to terminate the final string and another to terminate the array. The string pointer list s is terminated by a null pointer.

RETURNS

The return value indicates how many string pointers were placed into the array s, not including the NULL terminator If the number of strings plus the final null pointer is greater than max, a value of -1 is returned.

SEE

getfnl, strsrt

strcat, strncat

Concatenate strings

Class: ANSI *Category: String Copy*

SYNOPSIS

```
#include <string.h>

p = strcat(to,from);
p = strncat(to,from,n);

char *p;            same as destination string pointer
char *to;           destination string pointer
const char *from;   source string pointer
size_t n;           length count
```

DESCRIPTION

The strcat function concatenates the source string to the tail end of the destination string. Compare this function with strncat, which allows you to specify the maximum number of characters which will be added.

A null byte is placed at the end of the destination.

RETURNS

The strcat and strncat functions return a pointer that is the same as the first argument.

SEE

strcpy, stpcpy, strncpy

EXAMPLE

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

int main(void)
{
  char a[256],b[256];
  long n;

  for (;;)
  {
    printf("\nEnter string A: ");
    if(gets(a) == NULL)
      break;
    printf("Enter string B: ");
    if(gets(b) == NULL)
      break;
    printf("Enter maximum length N: ");
    scanf("%ld",&n);
    printf("strcat(A,B): \"%s\"\n",strcat(a,b));
    printf("strncat(A,B,N): \"%s\"\n",strncat(a,b,n));
  }
  return 0;
}
```

strchr, strrchr, stpchr, stpchrn Find character

Class: ANSI *Category: String Search*

SYNOPSIS

```
#include <string.h>

p = stpchr(s,c);      find character in string
p = stpchrn(s,c);     find character not in string
p = strchr(s,c);      find character in string
p = strrchr(s,c);     find last character in string

char *p;              updated string pointer
const char *s;        input string pointer
int c;                character to be located
```

DESCRIPTION

The stpchr and strchr functions scan the input string to find the first occurrence of the character specified by argument c. Similarly, stpchrn scans for the first occurrence of some character other than c. The strrchr function scans the input string to find the last occurrence of the character specified by argument c.

stpchr is provided for compatibility with other versions of Lattice C, whilst the strchr function is now part of the ANSI standard.

RETURNS

For strchr, strrchr and stpchr a NULL pointer is returned if the input string is empty or if the specified character is not found. stpchrn returns a NULL pointer if the input string is empty or consists entirely of character c.

strcmp, stricmp, strncmp, strnicmp

Compare strings

Class: ANSI *Category: String Comparison*

SYNOPSIS

```
#include <string.h>

x = strcmp(a,b);      Compare strings
x = stricmp(a,b);     Compare strings, case-
                      insensitive
x = strncmp(a,b,n);   Compare strings, length-limited
x = strnicmp(a,b,n);  Compare strings, no case, max
                      size

int x;                comparison result
const char *a,*b;     strings being compared
size_t n;             length limiter
```

DESCRIPTION

These functions compare two null-terminated strings. The ASCII collating sequence is used in all cases, but stricmp and strnicmp do not distinguish between upper and lower case. Note also that stricmp and strnicmp are not part of the ANSI C standard.

The relative collating sequence of the strings is indicated by the sign of the return value, as follows:

Return	Meaning
Negative	First string is below second
Zero	Strings are equal
Positive	First string is above second

If the strings have different lengths, the shorter one is treated as if it were extended with zeroes. For strncmp and strnicmp, no more than n characters are compared.

Note that strcmp has a built-in version which is functionally equivalent to the standard library version. The statement #include <string.h> provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement i.e. #undef strcmp.

RETURNS

As noted above.

EXAMPLE

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

void result(const char *name, size_t r)
{
  char *p;

  if(r == 0)
    p = "is equal to";
  else if(r < 0)
    p = "is less than";
  else if(r > 0)
    p = "is greater than";
  printf("%s String A %s string B\n",name,p);
}

int main(void)
{
  char a[256],b[256];
  long n;

  for (;;)
  {
    printf("Enter string A: ");
    if(gets(a) == NULL)
      break;
    printf("Enter string B: ");
    if(gets(b) == NULL)
      break;
    printf("Enter maximum compare length: ");
    scanf("%d",&n);

    result("strcmp:  ",strcmp(a,b));
    result("stricmp:  ",stricmp(a,b));
    result("strncmp:  ",strncmp(a,b,n));
    result("strnicmp:",strnicmp(a,b,n));
  }
  return 0;
}
```

Class: ANSI — *Category: String Comparison*

SYNOPSIS

```
#include <string.h>

num = strcoll(s1,s2);

int num;          integer indicating comparison result
const char *s1;   first string to be compared
const char *s2;   second string to be compared
```

DESCRIPTION

The strcoll function compares the string pointed to by s1 with the string pointed to by s2, with both interpreted as appropriate to the LC_COLLATE (defined in locale.h) category of the current locale.

The strcoll and strxfrm functions provide locale-specific string sorting. The former is intended for applications in which the number of comparisons is small, while the latter is more appropriate when items are to be compared a number of times; the cost of transformation is then only paid once.

RETURNS

The strcoll function returns an integer greater than, equal to, or less than zero, as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 when both are interpreted as appropriate to the current locale.

SEE

strxfrm

strcpy, strncpy, stccpy, stpcpy — Copy strings

Class: ANSI — *Category: String Copy*

SYNOPSIS

```
#include <string.h>

p = strcpy(to,from);
p = strncpy(to,from,n);
size = stccpy(to,from,n);
np = stpcpy(to,from);

char *np;           points to end of destination
                    string
char *p;            same as destination pointer
char *to;           destination pointer
const char *from;   source pointer
size_t n;           maximum source length
size_t size;        number of bytes copied
```

DESCRIPTION

These functions copy the null-terminated source string to the destination area. For stpcpy and strcpy, the entire source string is copied, and the resulting destination is always null-terminated. The strncpy function always writes *exactly* n characters to the destination. If the null terminator is hit before n characters are copied from the source, then the destination is filled with null bytes. If the source string contains more than n non-null characters, the destination will not be null-terminated.

The stccpy function is similar to strncpy except that it always produces a null-terminated string, and it returns the actual number of bytes (size) placed in the to area, including the null terminator. Note that it may copy less than n bytes.

Note that strcpy has a built-in version which is functionally equivalent to the standard library version. The statement #include <string.h> provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement i.e. #undef strcpy.

Note that stpcpy and stccpy do not form part of the ANSI C standard, also note that you should be careful when using strncpy, since it is one of the few string functions which does not produce a null-terminated string under every condition.

RETURNS

The strcpy and strncpy functions return a pointer that is the same as the destination pointer. The Lattice function stpcpy returns a pointer to the end of the destination string, which is often more useful when you are building a string up from several pieces.

EXAMPLE

```
/*
 * This example should print: Hello, my name is John.
 *
 */
#include <string.h>

int main(void)
{
  char b[256],*p;

  p = stpcpy(b,"Hello, ");
  p = stpcpy(p,"my name is ");
  p = stpcpy(p,"John.");
  puts(b);
  return 0;
}
```

strdup

Duplicate a string

Class: XENIX *Category: String Copy*

SYNOPSIS

```
#include <string.h>

p = strdup(s);

char *p;            points to duplicate string
const char *s;      points to string being duplicated
```

DESCRIPTION

This function creates a duplicate of the specified string by using malloc and strcpy to allocate space and copy the string to it.

RETURNS

A NULL pointer is returned if malloc fails. Otherwise, the function returns a pointer to the duplicate string.

strerror — Map error number in errnum to error message

Class: ANSI *Category: Errors*

SYNOPSIS

```
#include <string.h>

errmsg = strerror(errnum);

char *errmsg;     error message string
int errnum;       error number
```

DESCRIPTION

The strerror function maps the value in errnum to an error message string pointed to by errmsg.

RETURNS

The strerror function returns a pointer to the string, the contents of which is an error message. The array pointed to cannot be modified by the program, but may be overwritten by a subsequent call to the strerror function.

strftime — Format using locale control parameters

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

ret = strftime(s,maxsize,format,timeptr);

size_t ret;              0 if successful
char *s;                 array to contain characters
size_t maxsize;          maximum number of characters
const char *format;      specifier to control formatting
const struct tm *timeptr;
                         time values
```

DESCRIPTION

The strftime function places characters into the array pointed to by s as controlled by the string pointed to by format. The format is a multibyte character sequence, beginning and ending in its initial shift state. The format string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behaviour of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. No more than maxsize characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described later. The appropriate characters are determined by the LC_TIME category of the current locale and by the values contained in the structure pointed to by timeptr.

The strftime function provides a way of formatting the date and time in the appropriate locale-specific fashion, using the %c, %x, and %X format specifiers. More generally, it allows the programmer to tailor whatever date and time format is appropriate for a given application. The facility is based on the UNIX system date command, by which each conversion specifier is replaced by appropriate characters described in the following list:

Code	Replaced by
%a	the locale's abbreviated weekday name
%A	the locale's full weekday name
%b	the locale's abbreviated month name
%B	the locale's full month name
%c	the locale's appropriate date and time representation

%d	the day of the month as a decimal number (01-31)
%H	the hour (24-hour clock) as a decimal number (00-23)
%I	the hour (12-hour clock) as a decimal number (01-12)
%j	the day of the year as a decimal number (001-366)
%m	the day of the month as a decimal number (01-31)
%M	the minute as a decimal number (00-59)
%p	the locale's equivalent of the AM/PM designations associated with a 12-hour clock
%S	the second as a decimal number (00-61)
%U	the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00-53)
%w	the weekday as a decimal number with Sunday as 0 (0-6)
%W	the week number of the year (the first Monday as the first day of week 1) as a decimal number (00-53)
%x	the locale's appropriate date representation
%X	the locale's appropriate time representation
%y	the year without century as a decimal number (00-99)
%Y	the year with century as a decimal number
%Z	the time zone name or abbreviation, or by no characters if no time zone is determinable
%%	two % characters are required to specify a single %

RETURNS

If the total number of resulting characters including the terminating null character is not more than maxsize, the strftime function returns the number of characters placed into the array pointed to by s not including the terminating null character. Otherwise, zero is returned and the contents of the array are truncated to maxsize characters and will not be null ('\0') terminated.

strins

Insert a string

Class: Lattice *Category: String Copy*

SYNOPSIS

```
#include <string.h>

strins(to,from);

char *to;               destination string
const char *from;       source string
```

DESCRIPTION

This function inserts the source string (to) in front of the destination string (from). Both strings must be null-terminated, and the destination is shifted to the right (upward in memory) in order to accomodate the source string. The final result is a single null-terminated string.

SEE

strcat

EXAMPLE

```
#include <string.h>

char here[] = "Here ";
char now[30] = "and now";

strins(now,here);     /* now => "Here and now" */
```

strlen, stclen

Measure length of a string

Class: ANSI *Category: String Copy*

SYNOPSIS

```
#include  <string.h>

length  =  strlen(s);   Measure  length  of  a  string
length  =  stclen(s);   Measure  length  of  a  string

const  char  *s;
size_t  length;    number  of  bytes  in  s  (before  NULL)
```

DESCRIPTION

These functions return the number of bytes in string s before the null terminator byte. The strlen function is the ANSI equivalent of the Lattice implementation stclen.

Note that strlen has a built-in version which is functionally equivalent to the standard library version. The statement #include <string.h> provides a default setting by which built-in functions are accessed. If you don't want the built-in function, you can use an #undef statement as i.e. #undef strlen.

RETURNS

The number of bytes in the string s before the null byte.

strlwr, strupr

Change case of string

Class: XENIX *Category: String Conversion*

SYNOPSIS

```
#include <string.h>

p = strlwr(s);           convert string to lower case
p = strupr(s);           convert string to upper case

char *p;                 return pointer (same as s)
char *s;                 string pointer
```

DESCRIPTION

These functions convert all alphabetic characters in the specified null-terminated string to lower or upper case. In each case, the function return value is the same as the string pointer.

RETURNS

Both functions return the original string pointer.

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

strmfe(newname,oldname,ext);

char *newname;              new file name
const char *oldname;        old file name
const char *ext;            extension
```

DESCRIPTION

This function copies the old file name to the new name, deleting any extension. Then it appends the specified extension to the new file name, with an intervening period. For example,

Oldname	Ext	Newname
"c:myprog.c"	"cc"	"c:myprog.cc"
"abc"	"prg"	"abc.prg"

The `newname` area must be large enough to accept the file name string and the separator. A safe size is FMSIZE, which is defined in the `dos.h` header file.

SEE

strmfn, strmfp

strmfn

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

strmfn(file,drive,path,node,ext);

char *file;              file name pointer
const char *drive;       drive code pointer
const char *path;        directory path pointer
const char *node;        node pointer
const char *ext;         extension pointer
```

DESCRIPTION

This function makes a file name from four possible components. In general, the name is constructed as follows:

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

If the drive pointer is not NULL, that string is moved to the area pointed to by the file argument. Then a colon is inserted unless one is already there. Next, if path is not NULL, it is appended to file, and the directory separator specified by _SLASH is added if necessary. The node string is appended next, unless it is NULL. Finally, if ext is not NULL, a period is appended to file, followed by the ext string.

RETURNS

None. Make sure that the file pointer refers to an area that is large enough to hold the result. A safe value is FMSIZE, which is defined in dos.h.

SEE

strmfe, strmfp, _SLASH

strmfp

Make file name from path/node

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

strmfp(name,path,node);

char *name;           file name
const char *path;     directory path
const char *node;     node
```

DESCRIPTION

This function copies the path string to the file name area, appending the _SLASH separator if the path string is not empty and does not end with a slash, backslash, or colon. Then the node string is appended to the file name. _SLASH is an external character variable that defaults to a backslash (\).

The name area must be large enough to accept the file name string. A safe value is FMSIZE, which is defined in the dos.h header file.

SEE

strmfe, strmfn, _SLASH

strmid

Return a substring from a string

Class: Lattice *Category: String Copy*

SYNOPSIS

```
#include <string.h>

error = strmid(source,dest,pos,len);

char *dest;                 destination pointer
const char *source;         source pointer
size_t pos;                 starting position of dest in
                            source
size_t len;                 length of substring
int error;                  -1 if pos is beyond source,
                            else 0
```

DESCRIPTION

The strmid function returns a pointer to a substring of source beginning at character position pos, and having a length of len. If len is greater than the length of source offset at pos, then the rest of the string is copied to dest.

The destination string is null-terminated.

RETURNS

If pos is beyond the length of source, then -1 is returned. Otherwise, 0 is returned.

SEE

strins

strpbrk,stpbrk

Find break character in string

Class: ANSI *Category: String Search*

SYNOPSIS

```
#include  <string.h>

p  =  stpbrk(s,b);
p  =  strpbrk(s,b);

char  *p;           points  to  break  character  in  s
const  char  *s;    string  to  be  scanned
const  char  *b;    break  characters
```

DESCRIPTION

These functions scan string s to find the first occurrence of a character from break string b. They are completely equivalent, except that strpbrk is the ANSI name, while stpbrk is the traditional Lattice name.

RETURNS

If no character from b is found in s, a NULL pointer is returned. Otherwise, p is a pointer to the break first break character.

SEE

strspn, strcspn

EXAMPLE

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

/*
 *  Scan  for  commas,  periods,  and  blanks.  Display  the
 *  tail  of  the  string  each  time  a  break  character  is
 *  found.
*/
char  *p,s[  ]  =  "Hello,  I  must  be  going.";

for(p  =  s;  p  =  strbrk(p,",.  ");  )
  printf("%s\n",p);
```

strrev

Reverse a character string

Class: XENIX *Category: String Copy*

SYNOPSIS

```
#include <string.h>

p = strrev(s);

char *p,*s; string pointer
```

DESCRIPTION

This function reverses a character string. That is, it "reflects" the string about its mid-point such that the last character is first and the first is last.

RETURNS

This function returns the same pointer that was passed to it.

EXAMPLE

```
char *s="Rotavator";

printf("%s reversed is ",s);
strrev(s);
printf("%s\n",s);

/* will print "Rotavator reversed is rotavatoR" */
```

strset, strnset

Set string to value

Class: XENIX *Category: String Copy*

SYNOPSIS

```
#include <string.h>

p = strset(s,c);
p = strnset(s,c,n);

char *p;        return pointer (same as s)
char *s;        string pointer
int c;          value
size_t n;       maximum string length
```

DESCRIPTION

The strset and strnset functions set all bytes of a null-terminated string to the same value, not including the terminator byte. With the strnset function, you can specify a maximum length in bytes, given by n.

RETURNS

The original string pointer is returned.

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

strsfn(file,drive,path,node,ext);

const char *file; file name pointer
char *drive;      drive code pointer
char *path;       directory path pointer
char *node;       node pointer
char *ext;        extension pointer
```

DESCRIPTION

This function splits a file name into four possible components and places them into the drive, path, node, and ext strings. If any of those arguments are NULL, then those components are discarded.

In general, a complete file name is constructed as follows:

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

When strsfn splits the file name, it leaves the colon attached to the drive code, but removes trailing punctuation from the other components. Slashes or backslashes within the path component are preserved. If the file name is of the form

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

then the path component is a single backslash.

RETURNS

You must make sure that the drive, path, node, and ext pointer refer to areas that are large enough to hold the largest string that might be generated. The following lengths are safe:

Part	Size
drive	3 bytes
path	FMSIZE in dos.h
node	FNSIZE in dos.h
ext	FESIZE in dos.h

This function does not check that these lengths are not exceeded, although it does copy file string to an internal buffer of size FMSIZE and truncate it if it is too long. If you want to be absolutely sure that no overflows occur, make each component area be FMSIZE bytes long.

SEE

strgfn, strmfe, strmfn

EXAMPLE

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

char  a[3],b[FMSIZE],c[FNSIZE],d[FESIZE];

/*
 *  After the next statement, the component strings
 *  are:
 *
 *  a  =>  ""
 *  b  =>  "abc\\def"
 *  c  =>  "ghi"
 *  d  =>  ""
 */

  strsfn("abc\\def\\ghi",a,b,c,d);

/*
 *  After the next statement, the component strings
 *  are:
 *  a  =>  "b:"
 *  b  =>  ""
 *  c  =>  "myfile"
 *  d  =>  "str"
*/

  strsfn("b:myfile.str",a,b,c,d);
```

strspn, strcspn, stcis, stcisn

Measure character span

Class: ANSI *Category: String Search*

SYNOPSIS

```
#include <string.h>

len = strspn(s,b);    Measure span of chars in set
len = strcspn(s,b);   Measure span of chars not in set
len = stcis(s,b);     Measure span of chars in set
len = stcisn(s,b);    Measure span of chars not in set

size_t len;         span length in bytes
const char *s;      points to string being scanned
const char *b;      points to character set string
```

DESCRIPTION

These functions measure the number of characters at the beginning of input string `s` that are either in or not in the character set specified by `b`. The `stcis` and `strspn` functions are identical and count the number of leading characters that are in the set. Similarly, `stcisn` and `strcspn` are identical and count the number of leading characters that are not in the set. The `stc` pair are provided for compatibility with other versions of Lattice C, while the `str` functions are now part of the ANSI standard.

RETURNS

The functions all return the number of bytes that are in or not in the specified character set. Note that the scan always stops when the null terminator byte is reached.

EXAMPLE

```
#include <stdio.h>
#include <string.h>
int main(void)
{
  char s1[256],s2[256];

  for(;;) {
    printf("\nEnter test string: ");
    if(gets(s1) == NULL) exit(0);
    printf("Enter span string: ");
    if(gets(s2) == NULL) exit(0);
    printf("strspn: %ld\n",(long)strspn(s1,s2));
    printf("strcspn: %ld\n",(long)strcspn(s1,s2));
    printf("stcis: %d\n",stcis(s1,s2));
    printf("stcisn: %d\n",stcisn(s1,s2));
  }
  return 0;
}
```

Class: Lattice *Category: Search and Sort*

SYNOPSIS

```
#include <string.h>

strsrt(s,n);

char *s[];     string pointer list
size_t n;      number of pointers in list
```

DESCRIPTION

This function performs a simple insertion sort of the string pointers in the specified list. It is particularly useful in conjunction with the getfnl and strbpl functions. For large lists, you will usually get better performance using tqsort.

SEE

getfnl, strbpl, tqsort

EXAMPLE

```
/*
 * This program constructs an array of pointers to
 * all file names in the current directory that have
 * a ".c" extension. Then the array is sorted by
 * ASCII order.
 */

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

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

void foo(void)
{
  int count;

  count = getfnl("*.c",names,sizeof(names),0);
  if(count > 0)
  {
    if(strbpl(pointers,300,names) != count)
      break;
  strsrt(pointers,count);
  }
}
```

Class: ANSI *Category: String Search*

SYNOPSIS

```
#include <string.h>

ptr = strstr(s1,s2);

char *ptr;          pointer to substring in string
const char *s1;     string to be searched
const char *s2;     substring to locate
```

DESCRIPTION

The strstr function locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2.

RETURNS

The strstr function returns a pointer to the located string, or a NULL pointer if the string is not found. If s2 points to a string with zero length, the function returns s1.

strtod — Convert initial string portion to double

Class: ANSI — *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr);

double val;          converted value
const char *nptr;    string portion to be converted
char **endptr;       points to object containing
                     pointer to final string
```

DESCRIPTION

The strtod function converts the initial portion of the string pointed to by nptr to double representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the isspace function), a subject sequence resembling a floating-point constant; and a final string of one or more unrecognised characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point character, then an optional exponent part, but *no* floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal point character (whichever occurs first) is interpreted as a floating point constant, except that the decimal point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

The strtod and strtol functions have been adopted by ANSI (from UNIX System V) because they offer more control over the conversion process, and because they are not required to produce unexpected results on overflow during conversion.

RETURNS

The strtod function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus HUGE_VAL is returned (according to the sign of the value), and the value of the macro ERANGE is stored in errno. If the correct value would cause underflow, zero is returned and the value of the macro ERANGE is stored in errno. Upon coverting the first part of nptr, the strtod function returns a pointer to the first character that is not part of the number. The converted double is returned.

Class: ANSI *Category: String Search*

SYNOPSIS

```
#include <string.h>

t = strtok(s,b);

char *t;            token pointer
char *s;            input string pointer or NULL
const char *b;      break character string pointer
```

DESCRIPTION

This function treats the input string as a series of one or more tokens separated by one or more characters from the break string. By making a sequence of calls to strtok, you can obtain the tokens in left-to-right order. To get the first (leftmost) token, supply a non-NULL pointer for the s argument. Then to get the next tokens, call the function repeatedly with a NULL pointer for s, until you get a NULL return pointer to indicate that there are no more tokens. The break string can be changed from one call to another.

Each time it is entered, strtok takes the following steps:

- If the input string is NULL, obtain the string pointer that was used on the preceding call. Otherwise use the new input string pointer.
- Scan forward through the string to the next non-break character. If it is a null byte, return a value of NULL to indicate that there are no more tokens.
- Scan forward through the string to the next break character or the null terminator. In the former case, write a null byte into the string to terminate the token, and then scan forward until the next non-break is found. In either case, save the final value of the string pointer for the next call, and return the token pointer.

Note that the input string gets changed as the scan progresses. Specifically, a null byte is written at the end of each token.

RETURNS

A NULL pointer is returned when there are no more tokens.

SEE

stptok, strcspn, strspn

EXAMPLE

```
/*
 *
 * This example breaks out words that are separated
 * by blanks or commas. The token pointer takes on
 * the following values as the program loops:
 *
 * LOOP TOKEN
 * 1    "first"
 * 2    "second"
 * 3    "third"
 * 4    "fourth"
 * 5    NULL
 */

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

int main(void)
{
  char test[] = "first, second third, fourth";
  char *token;

  token = strtok(test,", ");
  while(token != NULL)
  {
    printf("%s\n",token);
    token = strtok(NULL,", ");
  }
  return 0;
}
```

Class: ANSI *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

r = strtol(p,np,base);

long int r;        result
const char *p;     input string pointer
char **np;         receives new input string pointer
int base;          conversion base
```

DESCRIPTION

This function converts an ASCII input string into a long integer according to the specified base, which can range from 0 to 36, excluding 1. Valid digit characters are 0 to 9, a to z, and A to Z. The highest allowable character is determined by the conversion base. For example, if the base is 17, then the string can contain digits from 0 to 9, a to g, and A to G.

The function skips leading white space and then checks for a leading plus or minus sign. In the latter case, the result of the conversion is negated before it is returned. The conversion stops at the first invalid character, and a pointer to that character is returned in np if the np argument is not NULL. Note that if the entire string is converted, np will contain a pointer to the null terminator byte.

If base is 0, the string is analysed to see if it is octal, decimal, or hexadecimal:

Base 16
: If the string begins with 0x or 0X, base 16 (hexadecimal) conversion is performed.

Base 8
: Otherwise, if the string begins with 0, base 8 (octal) conversion is performed.

Base 10
: If neither of the above applies, base 10 (decimal) conversion is performed.

RETURNS

The strtol function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, LONG_MAX is returned for overflow, or LONG_MIN for underflow. The value of the macro ERANGE is stored in errno.

SEE

atol, stcd_l, strtoul

EXAMPLE

```
/*
 * This program tests the strtol function.
 */

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

int main(void)
{
  char *p,buff[80];
  int base;
  long x;

  for (;;)
  {
    printf("\nEnter number base (0 to 36): ");
    if(gets(buff) == NULL)
      break;
    if(buff[0] == '\0')
      break;
    base = atoi(buff);
    if((base < 0) || (base > 36))
      continue;
    printf("Enter number: ");
    if(gets(buff) == NULL)
      break;
    if(buff[0] == '\0') exit(0);
    x = strtol(buff,&p,base);
    printf("Decimal result = %ld\n",x);
    if(*p != '\0')
      printf("Residual = %s\n",p);
  }
  return 0;
}
```

Class: ANSI — *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

val = strtoul(nptr,eptr,base);

unsigned long int val;    converted value
const char *nptr;         string portion to be
                          converted
int base;                 radix specifier
char **eptr;              points to object containing
                          pointer to final string
```

DESCRIPTION

The strtoul function converts the initial portion of the string pointed to by nptr to unsigned long int representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the isspace function), a subject sequence resembling an unsigned integer represented in some radix determined by base; and a final string of one or more unrecognised characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as an integer constant according to the ANSI syntax. If the subject sequence has the expected form and the value of base is between 2 and 37, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to endptr, provided that eptr is not a NULL pointer.

RETURNS

The strtoul function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, ULONG_MAX is returned, and the value of the macro ERANGE is stored in errno.

SEE

atol, stcd_l, strtol

Class: ANSI *Category: Localisation*

SYNOPSIS

```
#include <string.h>

len = strxfrm(s1,s2,n);

size_t len;       length of transformed string
char *s1;         array containing transformed string
const char *s2;   pointer to string to be transformed
size_t n;         maximum number of characters to
                  place
```

DESCRIPTION

The strxfrm function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the strcmp function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the strcoll function applied to the same two original strings. No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. If n is zero, s1 is permitted to be a NULL pointer. If copying takes place between objects that overlap, the behaviour is undefined.

The strcoll and strxfrm functions provide for locale-specific string sorting. The strcoll function is intended for applications in which the number of comparisons is small, while strxfrm is more appropriate when items are to be compared a number of times; the cost of transformation is then only paid once.

RETURNS

The strxfrm function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, the contents of the array pointed to by s1 will not be null ('\0') terminated.

EXAMPLE

```
/*
 * The value of the following expression is the size
 * of the array needed to hold the transformation of
 * the string pointed to by s.
 */

size_t len(const char *s)
{
  return 1 + strxfrm(NULL, s, 0);
}
```

stspfp

Parse file path

Class: Lattice *Category: File Name Manipulation*

SYNOPSIS

```
#include <string.h>

error = stspfp(path,nx);

int error;        -1 for error, 0 for success
char *path;       file path string
int nx[16];       node index array
```

DESCRIPTION

This function parses a file path, which is a null-terminated string consisting of nodes separated by the _SLASH character. Each separator is replaced with a null byte, and the index to the first character of that node is placed into the node index array. The last entry in the array is followed by a -1. A leading separator in the path string is skipped.

RETURNS

A return value of -1 indicates that the path contains more than 15 nodes.

SEE

stcgfe, stcgfn, stcgfp, strsfn, _SLASH

EXAMPLE

```
/*
 * The following parses \ABC\DE\F into strings ABC,
 * DE, and F. The node index array will then contain
 * 1, 5, 8, and -1.
 */

#include <string.h>

int xx[16];

stspfp("\\ABC\\DE\\F",xx);
```

_stub

Class: Lattice *Category: Process Environment*

SYNOPSIS

```
_stub();
```

DESCRIPTION

The _stub function is the default routine resolved by CLink for routines not found in libraries. By default, it will give a prompt indicating that the unwritten routine had been called. It is intended to allow development and testing of a program for which some of the routines have not been written (and, of course, are not expected to be called).

swab

Byte swap words

Class: UNIX *Category: Data Conversion/Formatting*

SYNOPSIS

```
#include <stdlib.h>

swab(src,dest,nbytes);

const void *src;     area to copy bytes from
void *dest;          area to copy bytes to
size_t nbytes;       number of bytes to exchange
```

DESCRIPTION

The swab function copies nbytes from src to dest, exchanging odd and even bytes as it does so. The value of nbytes should be even, also note that this function is undefined in the general overlapping block case (cf. memcpy), however when src==dest the function will perform as expected.

Note that this function is most often used when transferring data from one architecture to another (e.g. Intel - Motorola), where the order of bytes within words differs.

SEE

memmove, memcpy

Class: ANSI *Category: Process Creation*

SYNOPSIS

```
#include <stdlib.h>

error = system(cmd);

int error;                        non-zero if error
const char *cmd;                  command string

extern char *_comspecmagic;      "/c"
extern char *_shellmagic;        "-c"
```

DESCRIPTION

This function invokes the system command processor and passes the cmd string to it. The function will attempt to find a command processor by inspecting the _shell_p system variable, if this is non-NULL it will call through this vector with cmd as the sole argument.

If no resident shell can be found a command processor specified by SHELL or the COMSPEC environment variable is searched for, and so you must be sure that this variable is properly specified in your environment (if one is available). Under normal circumstances, you will automatically inherit a copy of this variable if your program starts. If neither of these exist (e.g. the program was run from the desktop), system will attempt to start a process using the forkl function.

When using the SHELL or COMSPEC environment variables many command processors require a command line switch to force them to accept a command on their command line, system makes the variables _shellmagic and _comspecmagic which have the default values "-c" and "/c" respectively. You may change these simply by moving the pointer to a new area of your own. Note that you should not copy into the old area as this has a strictly limited size.

If the cmd passed to system is NULL, then the return value specifies whether a command processor is available. Under GEMDOS this value will always be non-zero indicating that a command processor is available.

RETURNS

If the command processor cannot be invoked, a value of -1 is returned, and additional error information can be found in errno and _OSERR. Otherwise, the function returns the value that was passed back by the command processor.

SEE

errno, forkl, _OSERR

EXAMPLE

```
/*
 *  Run all the programs mentioned on the command
 *  line one after another
 */

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

int main(int argc, char *argv[]);
{
  while (--argc)
    system(*++argv);
  return 0;
}
```

Class: ANSI *Category: Date and Time*

SYNOPSIS

```
#include <time.h>

timeval = time(timeptr);

time_t timeval;  time value
time_t *timeptr; pointer to time value storage
```

DESCRIPTION

This function returns the current time expressed as the number of seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. If timeptr is not NULL, the time value is also stored in that location.

SEE

asctime, ctime, gmtime, localtime, _tzset, utpack, utunpk

EXAMPLE

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

int main(void)
{
  time_t t;

  time(&t);
  printf("Current time is %s\n",ctime(&t));

  return 0;
}
```

Class: UNIX *Category: Date and Time*

SYNOPSIS

```
extern int __daylight;     Daylight savings time flag
extern long __timezone;    Timezone bias from GMT
extern char *__tzname[2]; Timezone names
extern char __tzstn[4];    Standard time name
extern char __tzdtn[4];    Daylight time name
```

DESCRIPTION

These variables are initialised by the _tzset function and are used by the localtime function to adjust from Greenwich Mean Time (GMT) to the local time.

The __daylight item is non-zero if daylight saving time is currently in effect. The __timezone value is the number of seconds that must be subtracted from GMT. The two __tzname pointers point to __tzstn and __tzdtn, respectively. These strings contain the three-character names for standard time (__tzstn) and daylight time (__tzdtn).

SEE

localtime, _tzset

tmpfile

Create a temporary binary file

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

strm = tmpfile();

FILE *strm; pointer to file stream
```

DESCRIPTION

The tmpfile function creates a temporary binary file (mode "wb+") that will automatically be removed when it is closed or at program termination. If the program terminates abnormally the file may not be deleted correctly.

RETURNS

The tmpfile function returns a pointer to the stream of the file that it created. If the file cannot be created, the tmpfile function returns a NULL pointer.

SEE

fopen, mktemp, tmpnam

tmpnam

Create temporary file name

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

name = tmpnam(buff);

char *name;          points to file name
char *buff;          buffer for file name or NULL
```

DESCRIPTION

This function creates a unique file name and returns a pointer to the name.

If buff is not NULL, then the file name is placed in that buffer, and name will be the same as buff. The buffer must be large enough to hold the file name; L_tmpnam (defined in stdio.h) is a safe size.

If buff is NULL, then an internal buffer is used, and the function returns a pointer to it. Note that this internal buffer is changed on every call to tmpnam, even if buff is not NULL.

In previous releases, the file was created when the unique name was selected. In accordance with the ANSI standard, this no longer done.

RETURNS

A NULL return indicates that the unique file could not be created.

Class: ANSI *Category: Character Classification/Conversion*

SYNOPSIS

```
#include <ctype.h>

cc = toascii(c);      convert character to ASCII
cc = tolower(c);      convert character to lower case
cc = toupper(c);      convert character to upper case
cc = _tolower(c);     convert upper case character to
                      lower case
cc = _toupper(c);     convert lower case character to
                      upper case

int cc;               converted character
int c;                character to convert
```

DESCRIPTION

These functions convert characters into different forms. The toascii conversion simply resets all high-order bits, leaving only the lower seven. The tolower conversion tests if c is an upper case alphabetic character and, if so, converts it to lower case. Otherwise, cc is the same as c. The toupper conversion is the reverse of tolower.

The _toupper and _tolower functions perform a similar function to toupper and tolower, but do not check the case of the character before converting it. They are provided primarily for compatability with other systems and do *not* form part of the ANSI standard.

SEE

_ctype

EXAMPLE

```
/*
 * Echoe input lines in upper case.
 */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char b[100],*p;

  while(gets(b) != NULL) {
    for(p = b; *p != '\0'; p++)
      *p = toupper(*p);
      puts(b);
  }
  return 0;
}
```

Class: GEMDOS *Category: Process Environment*

SYNOPSIS

```
#include <dos.h>

extern short _tos;    major and minor OS version
```

DESCRIPTION

These variable gives the operating system version number, in the major/minor form used in the ROM. The high eight bits give the major version number (1 on all current releases), whilst the lower eight bits give the minor release number.

The currently used values are:

Major	Minor	Name
1	0	ROM TOS (1.0)
1	2	Blitter TOS (1.2)
1	4	Rainbow TOS (1.4)
1	6	STE TOS (1.6)

SEE

Sversion

EXAMPLE

```
/*
 * print out OS version number
 */

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

void show_version(void)
{
  printf("TOS version=%d.%d\n",_tos>>8,_tos&0xff);
}
```

Class: ANSI *Category: Mathematics*

SYNOPSIS

```
#include <math.h>

r = cos(x);          Cosine function
r = sin(x);          Sine function
r = tan(x);          Tangent function

r = acos(x);         Arccosine function
r = asin(x);         Arcsine function
r = atan(x);         Arctangent function
r = atan2(x,y);      Arctangent of x/y

r = cosh(x);         Hyperbolic cosine function
r = sinh(x);         Hyperbolic sine function
r = tanh(x);         Hyperbolic tangent function

double r;            result;
double x,y;          arguments
```

DESCRIPTION

The cos, sin, and tan routines compute the usual circular functions of angles expressed in radians.

The acos, asin, atan, and atan2 routines compute the inverse circular functions, returning angular values expressed in radians. Results are constrained as follows:

Function	Return Range
acos	0 to π
asin	$-\frac{\pi}{2}$ to $\frac{\pi}{2}$

Function	Return Range
atan	$-\frac{\pi}{2}$ to $\frac{\pi}{2}$
atan2	$-\frac{\pi}{2}$ to $\frac{\pi}{2}$

Since the tangent becomes very large for angles close to $\frac{\pi}{2}$, the atan2 function is often used to avoid computations with large numbers that might easily overflow. With atan2, you can express the large tangent value as a quotient of two more reasonable numbers.

The cosh, sinh, and tanh routines compute the normal hyperbolic functions.

SEE

matherr

_tzset

Set time zone variables

Class: XENIX *Category: Date and Time*

SYNOPSIS

```
#include  <time.h>

_tzset();

/*  These  symbols  are  defined  in  time.h:
 *
 *  extern  int  __daylight;
 *  extern  long  __timezone;
 *  extern  char  *__tzname[2];
 *  extern  char  __tzstn[4];
 *  extern  char  __tzdtn[4];
 */
```

DESCRIPTION

The _tzset function assigns values to the time zone variables __daylight, __timezone, and __tzname. These variables are then used by localtime and other functions to correct from Greenwich Mean Time (GMT) to local time.

The values for these variables are obtained from the environment variable named TZ having the following form

```
set  TZ=aaabbbccc
```

where aaa is the 3-letter abbreviation for the local standard time zone (e.g. CET), and bbb is a number from -23 to +24 indicating the value that is subtracted from GMT in order to obtain local standard time. Both aaa and bbb are required, but ccc is the abbreviation for the local daylight savings time zone (e.g. BST), and it should be present only if daylight savings time is currently in effect.

When _tzset is called, it first tries to locate TZ in the environment string array and uses the default string "GMT0" if TZ isn't found. Then __timezone is loaded with the number of seconds that must be subtracted from GMT in order to get the local time. Next __daylight is loaded with 0 if the ccc portion of TZ is absent and 1 if ccc is present. Then the aaa and ccc parts are copied to __tzstn and __tzdtn, respectively, with null terminators. Finally, __tzname(0) and __tzname(1) are loaded with pointers to __tzstn and __tzdtn respectively.

SEE

_timedata, localtime

ungetc

Push input character back

Class: ANSI *Category: Stream I/O*

SYNOPSIS

```
#include <stdio.h>

r = ungetc(c, fp);

int r;          return character or code
int c;          character to be pushed back
FILE *fp;       file pointer
```

DESCRIPTION

This function pushes a character back to the specified buffered input file. The character need not be the same as the one that was most recently read. However, before calling ungetc, you must have read at least one character via fgetc or one of the other buffered input functions. Also, you can only push back one character; if you call ungetc more than once between input functions, the results are undefined.

RETURNS

Normally ungetc returns the character that was pushed back. However, if the end-of-file has been hit or if no characters have been read yet, the value EOF is returned.

SEE

fgetc, fgets, getc, gets

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  int c;

  for (;;)
  {
    printf("Loop 1...\n");
    while((c = getchar()) != EOF)
      if(isalpha(c))
        putchar(c);
      else
        break;
    ungetc(c);
  }
  printf("\n\nDone\n");
  return 0;
}
```

ungetch

Unget console keyboard character

Class: Lattice *Category: Console and Port I/O*

SYNOPSIS

```
#include <dos.h>

r = ungetch(c);

int c;
int r;
```

DESCRIPTION

The ungetch function is one of a group of functions that perform I/O operations with the keyboard and display attached as the console device.

The ungetch function pushes a character onto a stack so that it will be read on the next call to getch or getche. Also, kbhit will report that a character is waiting if one has been pushed onto the stack. The stack is only one level deep, and if you try to push a second character, the function will return -1 and ignore the request. Otherwise, it returns the character that was pushed. Also, note that if you push back a non-ASCII scan code, the next call to getch or getche will not produce the usual zero return to indicate that a scan code is coming. You can clear the stack by calling ungetch with a character value of 0.

RETURNS

As noted above.

SEE

cgets, cputs, getch, getche, kbhit, putch

utime

Set file modification time

Class: UNIX *Category: Low-Level I/O*

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>

err = utime(name,time)

int err;                    error return
const char *name;           name of file to manipulate
struct utimbuf *time;       time buffer
```

DESCRIPTION

The utime function changes the last modified time of the file name. If the value of time is NULL, then the modification time is set to the current time, if it is not NULL then it should point to utimbuf structure which has the following elements:

```
struct utimbuf
{
  time_t actime;      /* access time - ignored */
  time_t modtime;     /* new last modification time */
};
```

The modification time that is required should be placed in the modtime element of the structure.

RETURNS

The function returns 0 on succesfully changing the time of the file, or -1 to indicate an error, with further information in errno.

SEE

Fdatime, stat, time

utpack, utunpk — Pack or unpack UNIX time

Class: Lattice — *Category: Date and Time*

SYNOPSIS

```
#include <stdlib.h>

ut = utpack(x);        Pack UNIX time
utunpk(ut,x);          Unpack UNIX time

long ut;               packed UNIX time
char *x;               unpacked UNIX time
```

DESCRIPTION

These functions pack and unpack the 32-bit value time that is traditionally used in UNIX systems. This value is the number of seconds since 00:00:00, January 1, 1970. The time function returns the system clock in this form relative to Greenwich Mean Time.

The unpacked time is a 6-byte array in the following format:

Byte	Contents
x(0)	year - 1970 (-128 to +127)
x(1)	month (1 to 12)
x(2)	day (1 to 31)
x(3)	hour (0 to 23)
x(4)	minute (0 to 59)
x(5)	second (0 to 59)

Although this array is similar to the one produced by getclk and used by stpdate, note that the year is biased relative to 1970 instead of 1980. So, if you use utunpk followed by stpdate, you must subtract 10 from x(0) before the stpdate call. Note also that the year is a signed character and can be negative. A value of -3, for example, is 1967 (i.e. 1970 - 3).

SEE

ctime, getclk, gmtime, localtime, stpdate, time

EXAMPLE

```
/*
 * Get a file time and convert it to UNIX time.
 * No error checks.
 */

#include <time.h>
#include <dos.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
  char tt[6];
  int fh;
  long ft,ut;

  ft = getft(argv[1]);
  ftunpk(ft,tt);
  tt[0] += 10;
  ut = utpack(tt);
  printf("File time is: %s\n",ctime(&ut));
  return 0;
}
```

vfprintf — Formatted print to file, variable argument list

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

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

length = vfprintf(fp,fmt,arg);

int length;            number of characters generated
FILE *fp;              file pointer
const char *fmt;       format string
va_list arg;           variable argument list
```

DESCRIPTION

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by arg, which has been initialised by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro.

RETURNS

The vfprintf function returns the number of characters transmitted, or a negative value if an output error occurred.

SEE

printf, vprintf, vsprintf

EXAMPLE

```
/*
 * generalised error handler
 */

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

void error(const char *s,...)
{
  va_list args;

  fputs("Error:  ",stderr);
  va_start(args, s);
  vfprintf(stderr, s, args);
  va_end(args);
  fputc('\n',stderr);
  exit(EXIT_FAILURE);
}
```

vprintf

Formatted print to stdout, variable argument list

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

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

length = vprintf(fmt,arg);

int length;          number of characters generated
const char *fmt;     format string
va_list arg;         variable argument list
```

DESCRIPTION

The vprintf function is equivalent to printf, with the variable argument list replaced by arg, which has been initialised by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro.

RETURNS

The vprintf function returns the number of characters transmitted, or a negative value if an output error occurred.

SEE

printf, vfprintf, vsprintf

vsprintf

Formatted print to storage, variable argument list

Class: ANSI *Category: Formatted I/O*

SYNOPSIS

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

length  =  vsprintf(s,fmt,arg);

int  length;            number  of  characters  generated
char  *s;               storage  pointer
const  char  *fmt;      format  string
va_list  arg;           variable  argument  list
```

DESCRIPTION

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by arg, which has been initialised by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro. If copying takes place between objects that overlap, the behaviour is undefined.

RETURNS

The vsprintf function returns the number of characters written in the array, not counting the terminating null character.

SEE

printf, vfprintf, vprintf

wait

Wait for child process to complete

Class: UNIX *Category: Process Creation*

SYNOPSIS

```
#include <stdlib.h>

cc = wait();

int cc; completion code
```

DESCRIPTION

The wait function is used in conjunction with the fork functions, which 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. The wait function must be called to obtain the child process's completion code.

RETURNS

If the specified program file cannot be found using the fork function, 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, fork

wcstombs

Multibyte string conversion

Class: ANSI *Category: Wide Characters*

SYNOPSIS

```
#include <stdlib.h>

num = wcstombs(s,pwcs,n);

size_t num;               number of bytes modified
char *s;                  string to be converted
const wchar_t *pwcs;      array to store codes
size_t n;                 maximum number of bytes to be
                          modified
```

DESCRIPTION

The wcstombs function converts a sequence of codes that correspond to multibyte characters from the array pointed to by pwcs into a sequence of multibyte characters that begins in the initial shift state. It then stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb function, except that the shift state of the wctomb function is not affected.

No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behaviour is undefined.

RETURNS

If a code is encountered that does not correspond to a valid multibyte character, the wcstombs function returns ((size_t)-1). Otherwise the wcstombs function returns the number of bytes modified, not including a terminating null character, if any.

SEE

wctomb

wctomb — Determine bytes needed to represent multibyte character

Class: ANSI *Category: Wide Characters*

SYNOPSIS

```
#include <stdlib.h>

ret = wctomb(s,wchar);

int ret;
char *s;           array object in which character is
                   stored
wchar_t wchar;     multibyte character code value
```

DESCRIPTION

The wctomb function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is wchar (including any change in shift state). It stores the multibyte character representation in the array object pointed to by s (if s is not a NULL pointer). At most MB_CUR_MAX characters are stored. If the value of wchar is zero, the wctomb function is left in the initial shift state.

RETURNS

If s is a NULL pointer, the wctomb function returns a non-zero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a NULL pointer, the wctomb function returns -1 if the value of wchar does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of wchar.

In no case will the value returned be greater than the value of the MB_CUR_MAX macro.

SEE

wcstombs

_xcovf

Stack overflow exit

Class: GEMDOS *Category: Errors*

SYNOPSIS

```
_xcovf();
```

DESCRIPTION

This error exit is called whenever a potential stack overflow is detected by the function prologue. In other words, if the stack does not contain enough space to handle the needs of a function, _xcovf will be called when that function is activated. The default version prints a stack overflow message on the screen and aborts with exit code 3. We supply the source code for this version so you can change it for your particular application.

Note that any user supplied function must be compiled with stack checks off, otherwise the function will recursively call itself!

SEE

_base, _STACK, _STKDELTA

Index