getreg, putreg

Manipulate 68000-specific registers

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 iabs 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

is...

Character tests

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

lprintf

Formatted print to stdprt

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

lsbrk — Allocate a large block from linear heap

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);
}
```

main

Your main program

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

matherr, except

Math error handler

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

max, min

Compute maximum and minimum

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.

mblen — Determine number of bytes of multibyte character

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.

mem... — Memory block operations

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.

mkdir

Make a new directory

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

mktime

Convert to calendar time value

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]);
```

modf

Split floating point value

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");
}
```

onexit

Exit trap

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:

O_RDONLY	Read-only access. No writes are allowed.
O_WRONLY	Write-only access. No reads are allowed.
O_RDWR	Read-write access. Both reads and writes are allowed.
O_CREAT	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: Value — Meaning S_IWRITE — Write allowed S_IREAD — Read allowed S_IWRITE | S_IREAD — Both allowed 0 — Both allowed 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.
O_APPEND	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.