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 _lomode 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

opene — Open with environment search

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

_OSERR — GEMDOS Error Information

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

_pbase

Basepage of program

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

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

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);
  }
}
```


poserr

Print GEMDOS error message

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

printf

Formatted print to stdout

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 */
```


puts

Put a string to stdout

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);
}
```

raise — Send signal

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

rand — Generate random numbers

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

readdir

Read next directory entry

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.

rmdir — Remove a directory

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

sbrk — Allocate a short block from linear heap

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

scanf

Formatted input from stdin

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

sprintf — Formatted print to storage

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);
  }
}
```

stcd_i, et al — Convert strings to integer

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;
}
```

stci_d, et al

Convert integers to strings

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 stcl_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);
  }
}
```