The chain Function

===

Purpose:

Loads a CP/M file of arbitrary contents at a given address and then jumps to that address. The file control block area used for the call must be 100 bytes in length and not overlap the space to be used by the newly loaded program. Not a beginners function.

Function Header:

```
VOID chain(tpa,program)
char *tpa                       Load address for program.
char *fcb;                      File Control Block, 100 bytes long.
                                Safe area for the FCB and program
                                loader.
```

WARNING: The area at fcb must be 100 bytes in length and remain protected during the load of the program. You must not chain to a program that can over-write the controlling file control block and the program loader area.

Returns:

Nothing. The routine chain() returns to the caller only in case the file is not found. Otherwise, the COM file is loaded and run. There is no default buffer processing.

Example: Load and run a program that resides above the BIOS. It is assumed that addresses 0xE000 to 0xFFFF are unoccupied and that the file DRIVER.PRE has been assembled to load at 0xE100. The code at 0xE100 should do its thing and exit to warm boot.

```
#include <stdio.h>
main()
{
     makeFCB(0xE000,"DRIVER.PRE");
     chain(0xE100,0xE000);
     puts("DRIVER.PRE not found");
}
```

Notes:

- o The routine makeFCB() makes a CP/M standard file control block of 36 bytes from a string that is supposed to represent a file name.

- o The fcb area is an array fcb[100]. The first 36 bytes store the file control block for the file to be loaded. In the remaining 64 bytes is stored a copy of a relocatable loader program which reads the file off the disk into memory at the set TPA address. Clearly the area fcb[100] may not intersect the buffer area for the file read.

- o The action taken by chain() is to copy the file to the desired tpa address and then jump to the base in order to run the program.

- o There is no internal error-checking to see if the programmer is about to crash the system. Indeed, this routine is not for the faint of heart, for it will certainly sustain a few crashes during the debugging stages.

- o The ability to load a program anywhere in memory and run it might be seen as a kind of concurrency. It can be thought of as a way to initialize and load device drivers.

- o A convenient and safe place to select fcb[100] is the console buffer for C/80 programs. It is in high memory just under FDOS and 136 bytes in length. If the file is missing fails, then no harm comes to the program currently in memory.

The chdir Function

Purpose:

Change the working directory to a new drive and new user area.

Function Header:

```
chdir(s)
char *s;        String of format "A0:" for drive A: user 0.
                User areas are 0 to 15 and drives are A: to P:.
```

The drive is not changed unless it is present. The user area is not changed unless present.

Returns:

Nothing useful. If the drive does not mount, then this function might drop the user out to CP/M.

Example: Change the default drive to B: and the user area to 1.

```
#include <unix.h>
#include <stdio.h>
main()
{
  chdir("B1");
  puts("User 1, default drive B:");
  system("DIR");
}
```

Example: Change to user 15 on the currently logged disk.

```
#include <unix.h>
#include <stdio.h>
main()
{
  chdir("15");
  puts("User 15");
  system("DIR");
}
```

Notes:

o This is for CP/M 2.2 only. Note the analog under PC-DOS.

The chmod Function

Purpose:

Change the protection of an existing file. The CP/M file attributes that can be updated are $R/W, $R/O, $DIR, $SYS.

Function Header:

```
int chmod(name,mode)
char *name;             Name of the file to access
int  mode;              Unix mode number, see below.
```

The effect of chmod() is to set the file flags of a CP/M file as follows:

CP/M MODE	UNIX MODE	UNIX I-NODE PATTERN
$R/W & $DIR	04777	-rwxrwxrwx
$R/O & $DIR	04555	-r-xr-xr-x
$R/O & $SYS	04444	-r--r--r--
$R/W & $SYS	04666	-rw-rw-rw-

Returns:

0	File exists and operation was successful.
-1	File not found, or directory update failed. [Write-protect tabs can cause failure]

Example: Look up the disk file "A:NAMES", set the file to $R/W and $DIR, open the file for append and add the name "Ritchie", close the file and set its file attributes to $R/O + $DIR.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
FILE *fp,*fopen();
  if(chmod("A:NAMES",04777) == 0) {
    fp = fopen("A:NAMES","a+");
    if(fp == (FILE *)0) { puts("File error"); exit();}
    fputs("Ritchie",fp);
    fclose(fp);
    chmod("A:NAMES",04555);
  }
  else {
    puts("File not found or disk write-protected");
  }
}
```

Notes:

- o CP/M considers a file to be:
  readable - exists in the disk directory
  writable - exists and has $R/W attribute
  executable - exists and has $DIR attribute

- o We map CP/M flags to permissions as follows:

$DIR	execute permission on
$SYS	execute permission off
$R/W	write permission on
$R/O	write permission off

- o Chmod() makes Unix source code compile without errors, even though it may not run properly.

- o The octal Unix modes are:

04000	Set user id on execution (suid)
02000	Set group id on execution (sgid)
01000	Save text in swap area after execution
00400	owner read permission
00200	owner write permission
00100	owner execute permission
00040	group read permission
00020	group write permission
00010	group execute permission
00004	all others read permission
00002	all others write permission
00001	all others execute permission

- o For example, 04755 means to set user id, all modes for the user but write permission is turned off for group and all others.

- o To turn off all action of chmod(), insert this code segment into code that accesses chmod():

```
setatt()
{
   return 0;
}
```

The CHmode Function
==

Purpose:

Switches between raw character mode and line mode with built-in editing functions.

Function Header:

```
VOID CHmode(n)
int n;                    Character mode with echo for n = 0.
                          Line mode for n = 1.
                          Character mode no echo for n = 2.
```

Returns:

Nothing useful.

Example: Get a character from the user with echo but no wait for a carriage return and linefeed.

```
#include <stdio.h>
main()
{
     printf("Yes or No? <Y/N>: ");
     CHmode(0);
     printf(toupper(getchar()) == 'Y' ? "\bYES\n" : "\bNO\n");
     CHmode(1);
}
```

Example: Get a character from the user with no echo. No wait for a carriage return and linefeed.

```
#include <stdio.h>
main()
{
     printf("Yes or No? <Y/N>: ");
     CHmode(2);
     printf(toupper(getchar()) == 'Y' ? "YES\n" : "NO\n");
     CHmode(1);
}
```

Notes:

o Uses external global Cmode = 0, 1, or 2 in the switching.

o The backspace method used above in the first example presumes that the terminal can backspace. It does not handle CR/LF answers. The second method works without terminal problems.

o A good tool for handling user options is:

```
        CHmode(2);
        while(index("YN",toupper(getchar())) == (char *)0)
             putchar('\7');       /* blow horn */
        CHmode(1);
```

This method has the advantage of filtering out all bad input and therefore reducing program bugs as the code is revised.

o A shorter version of the above code is

```
        while(index("YN",toupper(getbyte())) == (char *)0)
             putchar('\7');       /* blow horn */
```

o The change to character mode from line mode has its problems. Some CP/M systems do not have interrupt driven I/O. Function keys may not work as expected.

o The variable **CCtlCk** controls the number of times the console is re-tried for console input after one character is read. Its default value is 6. Change it to any number from 1 to 255, as follows:

```
extern char CCtlCk;
CCtlCk = 12;
```

If you have trouble with function keys, then this variable may have to be changed to a higher number. Especially true for slower terminals and CPU speeds under 4mhz.

o In Cmode <> 1, any character is acceptable, including NULL. The method used for character mode is direct BIOS with manual echo in Cmode = 0. We assume that the BIOS can be found by reading the address at 0x0001.

o In Cmode = 1, the usual CP/M editing functions are active, plus ctrl-P, ctrl-S and ctrl-C. In addition, ctrl-B will cause a jump to the ctrl-B processor CtlB (). A ctrl-C will cause an exit to CP/M, regardless of its entry position on the line.

The chown Function

===

Purpose:

Change the owner of an existing file.

Function Header:

```
int chown(name,owner,group)

char *name;             Ascii file name, CP/M conventions

int  owner,             Owner of the file, an integer known
                        to Unix via the login file.
     group;             Unix group number, an integer that
                        Unix looks up at login.
```

Both the owner number and the group number are ignored by CP/M.

Returns:

0	File exists
-1	File not found

Notes:

- o CP/M treats this call as a file look-up only. The user of a CP/M system is the owner.
- o Chown() makes Unix source code compile without errors, even though it may not run properly.
- o Unix calls getuid(), getgid() can look up the owner and group numbers, while setuid() and setgid() are capable of setting the numbers.

The clearerr Function

==

Purpose:

Clears a stream error. This function does nothing under C/80.

Function Header:

```
int clearerr(fp)
FILE *fp;             Open stream pointer.
```

Returns:

Nothing.

Example: Compile Unix code under C/80.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int x;
     while((x = getchar()) != EOF) {
          if(ferror(stdout)) break;
          putchar(x);
     }
     clearerr(stdout);
     fputs("Operation complete\n",stdout);
}
```

Notes:

- o clearerr() is a no-op under C/80 because no provision has been made to survive a disk output error.

The close Function

==

Purpose:

Flushes the file buffer to disk, writes the file control block to the disk directory and releases the file descriptor.

Function Header:

```
int  close(fd)
int  fd;                  File descriptor
```

Returns:

0	It worked
-1	It failed

Example:

Close a file that was opened by open() and written to by write().

```
#include <unix.h>
#include <stdio.h>
main()
{
int fd;
  fd = open("MYFILE",1);
  if(fd) {
    write(fd,"This is a test",14);
    close(fd);
  }
}
```

Notes:

o Requires a file descriptor as argument.

o Uses fclose() .

o The fflush() done by fclose() is a no-op in case no stream I/O was done.

The cmpi, cmpl, cmpf and cmps Functions

Purpose:

Compare integer, long integer, float and string data for qsort(). These are not library functions but rather source code segments ready to insert into your application. Often seen as code macros (C/80 can't use code macros).

Function source code:

```
int cmpi(n,m)
int *n,*m;                    Pointers to 16-bit integer data.
{ return (*n - *m); }

int cmpl(n,m)
long *n,*m;                   Pointers to 32-bit integer data.
{ return (*n - *m); }

int cmpf(n,m)
float *n,*m;                  Pointers to 32-bit float data.
{ return (*n - *m); }

int cmps(n,m)
char *n,*m;                   Pointers to strings, null-terminated.
{ return (strcmp(n,m)); }
```

Returns:

For numerical compares:

0	*n == *m	(same as strcmp)
< 0	*n < *m	
> 0	*n > *m	

For string compares, the return is the same as for strcmp(), which in turn is the same as the above, except for interpretation.

Example: Use qsort() to sort integer data.

```
#include <unix.h>
#include <stdio.h>
int q[10] = {12,1,3,-2,16,7,9,34,-90,10};
int p[10] = {12,1,3,-2,16,7,9,34,-90,10};
main()
{
int i;
  qsort(p,10,2,cmpi);
  for(i=0;i<10;++i) printf("%d. %d, %d\n",i,p[i],q[i]);
}
```

Program output

```
0. -90, 12
1. -2, 1
2. 1, 3
3. 3, -2
4. 7, 16
5. 9, 7
6. 10, 9
7. 12, 34
8. 16, -90
9. 34, 10
```

Notes:

- The pointer usage is required in qsort().

- To overcome unterminated string problems, use a special cmps():

```
int length = 10;
cmps(n,m)
char *n,*m;
{
int i;
   for(i=0;i<length;++i)
     if(n[i]-m[i]) return (n[i]-m[i]);
   return 0;
}
```

The problem that this special compare overcomes is presented by strings that are not null-terminated, but have a fixed length.

The comp Function

==

Purpose:

Compares two strings for a substring match beginning at the first character of the first string.

Function Header:

```
int comp(str1,str2)
char *str1,*str2;              Source strings, null-terminated.
```

Returns:

```
 0  if str1 extends str2 (str2 is a substring)
-1  if str2 is longer than str1
 J  if str2 mismatches str1 at position J
```

Example: Find the word SAILOR inside a sentence typed by the user.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i;
     if(argc <= 1) exit();
     for(i=0;argv[1][i] != '\0';++i) {
       if(comp(&argv[1][i],"SAILOR")==0) {
          printf("Word SAILOR found at index %d\n",i);
          return;
       }
     }
     printf("Word SAILOR not found in string\n");
}
```

Notes:

- o comp() is similar to the BASIC function instr(A$,B$). It takes two arguments.

- o The value of comp("abcd","abc") is 0, because str2 = "abc" is a substring of str1 = "abcd"

- o The value of comp("xabcd","abc") is 1 because the mismatch is at the first string position (1, not 0).

- o The value of comp("abcd","abcabc") is -1 because the second string is longer than the first string.

The conout and con Functions

==

Purpose:

Conout() prints multiple-argument characters to the console device. No re-direction. Used primarily as a debugging tool or as fast output in graphics routines.

Con() is the driver routine for conout. It prints one character and takes one argument.

Function Header:

```
int conout(c1,c2,...)
char c1,c2,...                Characters to be printed.

int con(c)
char c;                       Character to be printed.
```

Returns:

Nothing useful. The printing is done by a standard BDOS function.

Example: Print escape codes to the Z29 screen to enable inverse video graphics characters to be displayed.

```
#include <stdio.h>
main()
{
     conout(27,'p',27,'F');
     conout('g','r','a','p','h','i','c','s',' ','t','e','s','t');
     conout(27,'G',27,'q');
}
```

Notes:

o Very low overhead. Recommended for ROM applications where the essential functions of putchar() are required.

o The symbol conout() is defined in STDIO.H. The word conout is not a library global symbol. Without the #define information in STDIO.H, this function does not exist.

o The word CON is a global symbol in the library. It does not require #define information to be accessible.

o Not to be confused with putc() or putchar(), which allow re-direction. Conout() and con() work only with the console device.

The Copen_ Function

==

Purpose:

Opens streams stdin and stdout prior to the startup code which calls the function main().

Function Header:

```
int Copen_()
```

Returns:

```
0         Always.
```

Example: Making a new Copen_() to turn off re-direction.

```
Copen_()
{
     return 0;
}
```

Notes:

- o The provided Copen () routine opens files for streams stdin and stdout using descriptors fin and fout, which are of cast extern int. The file names are obtained from char *p_fin,*p_fout, which were loaded with the proper pointers by the routine Tokens().
- o The standard way to handle re-direction failures is to fall into the abort exit. This is the method presently used by Copen_() in the main library.
- o See also Cexit (), which closes stdin and stdout streams during the standard Cexit() routine.

The core Function

Purpose:

Obtains contiguous memory from the system and sets the upper bound of the program+data area. Aborts with an error message on failure.

Function Header:

```
char *core(n)
int n;                  Amount of memory requested. The maximum value
                        of n is 32767. See notes below.
```

Returns:

addr	on success, addr = base address of the contiguous area of memory n bytes in length.
system	Failure. Prints "Out of Memory" and warm boots.

Example: Get a buffer, fill it with nulls.

```
#include <stdio.h>
main()
{
char *p,*core();
     p = core(4096);          /* warm boot on failure */
     fillchr(p,'\0',4096);
}
```

Example: Read a file by re-direction into heap space, copying the heap address to an array of character pointers. Print to console when done.

```
#include <stdio.h>
main()
{
int i,n;
char *p[1024];
char s[129];
char *core();
     n = 0;
     while(getln(s,128) != EOF) {
          if(n >= 1024) break;
          strcpy(p[n++] = core(strlen(s)+1),s);
     }
     for(i=0;i<n;++i) puts(p[i]);
}
```

Notes:

- o This function will work with n an unsigned integer. There is no failure recovery, however. The penalty is warm boot.

The cos Function

==

Purpose:

Compute the cosine of real float value x. The value x is assumed in radians (not degrees).

Function Header:

```
float cos(x)              Cosine.
float x;                  Float value for computation.
```

Returns:

```
number          Answer between -1 and 1
                No error checking for large x
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float cos();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("cos(%f) = %f, errno = %d\n",x,cos(x),errno);
}
```

Notes:

o The method used is cos(x) = sin(PI/2 + x).

o Use deg() and rad() for conversions.

The cosh Function

Purpose:

Compute the hyperbolic cosine of real float value x.

Function Header:

float cosh(x)	Hyperbolic cosine $(e^{x} + e^{-x})/2$.
float x;	Positive, negative or zero argument.

Returns:

number	x in range
INF	x too large positive (see notes).
-INF	x too large negative (see notes).
	errno = ERANGE returned to flag error.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float cosh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("cosh(%f) = %f, errno = %d\n",x,cosh(x),errno);
}
```

Notes:

o Method used is cosh(x) = (exp(x) + exp(-x))/2.0.

o The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The creat, creata, creatb Functions

Purpose:

Add a new file to the disk directory. The file can be referenced by descriptor fd but not as a stream file.

Function Header:

```
int  creat(name,pmode)          Ascii create file
char *name;                     Null-terminated file name
int  pmode;                     Unix protection mode, ignored

int  creata(name,pmode)         Same as creat(), see unix.h
char *name;
int  pmode;

int  creatb(name,pmode)         Binary create file
char *name;
int  pmode;
```

Returns:

fd	File descriptor for the opened file
-1	It failed

Example:

Create a binary file and write a buffer of raw data.

```
#include <unix.h>
#include <stdio.h>
main()
{
char buf[128];
int fd;
  fd = creatb("JUNK",0);
  if(fd) {
    fillchr(buf,'\7',128);
    writeb(fd,buf,128);
    close(fd);
  }
}
```

Notes:

o creata() is update mode, ascii.

o creatb() is update mode binary.

o creat() is defined only in the unix.h header file.

The CtlB_ Interrupt Function

==

Purpose:

Run a user subroutine each time ctrl-B is intercepted. Often this routine exits to warm boot or sets an exit flag from a deeply buried sequence of function calls. May be used in conjunction with setjmp() and longjmp().

Function Header:

```
int CtlB ()             Invoke user subroutine manually.
char *CtTB;             Storage for user subroutine address.
                        This address is initially set to warm boot.
```

Returns:

Return value of the user subroutine.

Example: Set up a break condition in a subroutine. This routine sets up a function myctlb() to allow the user to break out of the I/O loop by typing ctrl-B.

```
static int flag = FALSE;

myctrlb()
{
     flag = TRUE;
}

int fileprint()
{
char *p;
extern char *CtlB;
     p = CtlB;
     CtlB = myctrlb;
     puts("Enter ctrl-B to quit");
     puts("Printing...");
     while(flag == 0) {
          if(g()) break;       /* print a line */
          CtlCk();             /* check ctrl-B   */
     }
     CtlB = p;
}
```

Notes:

- o The routine CtlCk() intercepts type-aheads and puts them into the type-ahead buffer at Cbuf. If Cmode == 1, then a break character test is performed. Ctrl-B runs CtlB_().

- o You can manually run CtlB_(). No one ever does this, however. The ctrl-B processor is a kind of limited interrupt, which is enabled by the programmer and checked only when it makes sense to check for a user interrupt.

- o CtlCk() is called before each disk access for a read or write. Therefore, CtlB_() may be called without the programmer's knowledge.

The CtlCk Function

Purpose:

Check for ctrl-C and ctrl-B at run time.

Function Header:

```
int CtlCk()
```

Returns:

In any case, available characters are added to the console buffer. In line mode (Cmode = 1) ctrl-C causes an abort exit through a_exit(), while ctrl-B causes a jump to function CtlB_(). In character mode (Cmode = 0 or 2) the abort keys are ignored and processing continues. The character is echoed in Cmode = 0.

Example: Check for ctrl-C while copying from stdin to stdout. Most common re-direction is disk for stdin and line printer for stdout.

```
#include <stdio.h>
main()
{
int x;
  while(TRUE) {
        x = getchar();
        if(x == EOF) break;
        CtlCk();
        putchar(x);
  }
}
```

Notes:

- The internal operation of CtlCk() is to call the keyboard status routine to see if the user typed a character. If so, then the character is read no-echo and added to the console buffer.

- It is common to call CtlCk() during character output to devices. During disk reads and writes, CtlCk() is called before each disk access, automatically (see WRITE.CC).

- In character mode, a call to CtlCk() is the same as adding available characters to the console buffer. No special action is taken for ctrl-C or ctrl-B.

The cvupper Function

Purpose:

Converts a string to upper case only, skipping over those characters that do not qualify for conversion.

Function Header:

```
    char *cvupper(s)
    char *s;                        String to convert, null-terminated.
```

Returns:

The string address s, with the characters of the string converted to upper case, as appropriate. Only lower case a-z are converted.

Example: Convert an input line to all upper case.

```
    #include <stdio.h>
    main()
    {
    char s[129];

         printf("Enter a line of text: ");
         gets(s);
         cvupper(s);
         printf("UPPERcase line:\n%s\n",s);
    }
```

Notes:

- o This function is unlikely to exist in other libraries, but it is easily written from toupper(). It is suggested that you bury the uppercase conversion in cvupper().
- o Cvupper() is an honest function, it is not a macro, and it has no side effects.

The C_cmln Function

Purpose:

Copies the default buffer at 0x80 to the C console buffer located at (char *)Cbuf. Also copies the running program name from the CCP buffer area, if possible.

Function Header:

```
VOID C_cmln()
```

Returns:

Nothing. Does a buffer copy.

Example: Making a new C_cmln() to disable command line copying.

```
C_cmln()
{
    return 0;
}
```

Notes:

- To copy the command line but leave out the copying of the running program name, use n=peekb(0x80) to find out the length of the default buffer and then use strncpy(Cbuf,0x81,1+n).
- One function of the current C_cmln() is to copy the name of currently running program from the CCP buffer. This may or may not be successful, especially if program chaining is being used. Also beware if the CCP size is not standard.

The decodF Function

==

Purpose:

Decodes a file control block into a printable ASCII string.

Function Header:

```
char *decodF(fcb)
char fcb[13];           File control block section.
```

Returns:

```
(char *)p           p = address of a static area in memory where
                    the decoded string is located.
```

Example: Print out the default file control blocks at 0x5C, 0x6C.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
char *decodF();
  insert("FILE1.TXT FILE2.TXT");
  printf("File name from FCB at 0x5C = %s\n",decodF(0x5C));
  printf("File name from FCB at 0x6C = %s\n",decodF(0x6C));
}
```

Notes:

o The buffer for decodF() is re-used on each call. If you need to keep the decoded name around for a while, then copy it to safety.

o Only the first 13 characters of the file control block are accessed during the call.

o There is no bdos() or bios() call that simulates this function.

The deg Function

==

Purpose:

Changes float radians x to degrees.

Function Header:

```
float deg(x)              Conversion to degrees.
float x;                  Value in radians to convert.
```

Returns:

angle in degrees (a FLOAT)

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
float atof();
float deg();
     printf("Enter float x in radians: "); gets(s); x = atof(s);
     printf("deg(%f) = %f\n",x,deg(x));
}
```

Notes:

o Method used is angle = (180.0/PI)*x.

o No error codes returned. No need to code for errno.

o No overflow check.

o No auto conversion to FLOAT.

The dsort Function

Purpose:

Distribution sort in High-Speed Assembler. Implements Donald Knuth's MathSort algorithm for a single digit. Multiple calls are required to complete the sorting. Best for byte, integer, long and float data types. Stable.

Function Header:

```
struct mathsort {
int
size,         integer number of records to sort
offset;       integer offset in bytes to target byte (+ or -)
char
*input,       address of input table
*output,      address of output table
*count;       address of counter table
};

VOID dsort(table)
struct mathsort *table;
```

Returns:

Nothing. The output table is changed to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of long integers.

```
#define pfLONG 1
#include <stdio.h>
main()
{
int i;
char *tmp;
static
struct mathsort table;
auto
char *data_in[10],*data_out[10];
auto
int scratch[256];
static
long ldata[] = {
     9349324,213232,343243,655666,546554,
     3454445,454544,565421,324221,344343
};
long peekl();

     for(i=0;i<10;++i) data_in[i] = &ldata[i];
     table.input = data_in;
     table.output = data_out;
     table.count = scratch;
     table.size = 10;
     table.offset = 0;

     for(i=0;i<4;++i) {
     dsort(&table);                     /* sort one digit   */
     tmp = table.input;                 /* swap the input   */
     table.input = table.output;        /* and output       */
     table.output = tmp;                /* tables, then     */
     ++table.offset;                    /* increment offset */
     }

     for(i=0;i<10;++i) printf("%ld\n",peekl(data_in[i]));
}
```

Notes:

- o The order of the arguments is essential.

- o The **input table** is an array of addresses of strings, which need not be null-delimited. Generally, the strings are numbers. The algorithm is stable, permitting sorts of byte, integer, long, float and double data types. The **output table** is the sorted order of the input table.

- o Only one digit is processed on each call. To process a number of digits, write a loop which swaps the input and output tables. See the example above for method.

The dsort16 Function

==

Purpose:

Distribution sort for 16-bit integers in High-Speed Assembler. Implements Donald Knuth's Algorithm D (MathSort).

Function Header:

```
struct mathsort {
int
size,       integer number of records to sort
offset;     integer flag, presently ignored. See notes.
char
*input,     address of input table
*output,    address of output table
*count;     address of counter table
};

VOID dsort16(table)
struct mathsort *table;
```

Returns:

Nothing. The input table is changed to reflect the new sorted order. The actual data in memory is unchanged. The output table and the counter table return no information.

Example: Sort a list of integers.

```
#include <stdio.h>
main()
{
int i;
static struct mathsort table;
auto char *data_in[10],*data_out[10];
auto int scratch[256];
static int data[] = {
     9324,3232,3243,5666,6554,
     4445,4544,5421,4221,4343
};
unsigned peekw();
     for(i=0;i<10;++i) data_in[i] = &data[i];
     table.input = data_in;
     table.output = data_out;
     table.count = scratch;
     table.size = 10;
     dsort16(&table);
     for(i=0;i<10;++i) printf("%u\n",peekw(data_in[i]));
}
```

Notes:

- o The order of the structure arguments is essential. The routine accepts the ADDRESS of a structure table.
- o The MathSort algorithm is stable. In particular, duplicates are not moved relative to one another.
- o The **input table** is an array of addresses of integer data. The **output table** is the sorted order of the input table. Since two passes and a swap of in and out tables occurs, the sorted order appears in the input table when the routine dsort16() returns.
- o The actual integer data is elsewhere in memory, since the **input** array contains just the addresses of the data, not the data itself. On return, the **input** array contains the addresses of the integer data, in sorted order. The counter table and the output table are used as scratch space, hence contain nothing useful.
- o To sort 32-bit data with Dsort16(), use it once to sort the first 16 bits, then increment all the input table pointers by 2 and call Dsort16() again. Decrement all the input table pointers by 2, and the input table gives the sorted order.
- o Source code is in 8080 assembler. Algorithm in C is given in the source code comments. See the source archives.

The edata, end and etext Variables

Purpose:

Defines end, etext, edata locations in a C program. These are external character pointer variables.

Function Header:

```
extern char *end, *etext, *edata;
```

Example: Print the data locations in a C program.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern char *end, *etext, *edata;
  printf("end=%u, etext=%u, edata=%u\n",end,etext,edata);
}
```

Notes:

- end — The next usable address after program load. Marks the end of the file buffer area. This address is returned by sbrk(0).

- etext — The physical end of the program text before the file buffers and FCB buffers begin. The address contained here is used in IOTABLE.CC to define the file buffers. See the archives.

- edata — Same as etext for C/80 because code and data are in the same place.

The errno Variable

==

Purpose:

Error return variable for the transcendental functions. Also used for system error returns.

Function Header:

```
     extern int errno;         This variable is a global in CLIB.REL
```

Returns:

The system error number.

Example: Test for an error return after using the sine function.

```
     #include <math.h>
     #include <stdio.h>
     main()
     {
     static float f = 1.0123;
     float sin();
     extern int errno;
          errno = 0;
          printf("sin(%f) = %f, errno = %d\n",f,sin(f),errno);
     }
```

Notes:

- o Use perror() to print an error string for errno.
- o The variable in CLIB.REL will be used if you don't use the symbol errno in your program.
- o A compile error will occur if you don't declare errno as extern or as a global variable (outside any function and not static).

The error Function

Purpose:

Prints multiple-argument strings to the console device. No redirection. Used primarily as a debugging tool.

Function Header:

```
VOID error(str1,str2,...)
char *str1,*str2...          Source strings to be printed.
```

Returns:

Nothing useful. The printing is done by a direct console function that acts through BDOS.

Example: Print error messages in a program that will be burned into EPROM at a later date. Uses setjmp() and longjmp() to break out of deeply buried control loops.

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf  env[1];
main()
{
   switch(setjmp(env)) {
   case 0: break;
   case 1: error("Error 1 from InitSystem\n"); break;
   case 2: error("Error 2 from WakePorts"); break;
   case 3: error("Error 3 from MainLoop\n","Dead solenoid\n");
           break;
   case 4: error("Error 4 from MainLoop\n","Bad timing\n"); break;
   case 5: error("Error 5 from MainLoop\n","Bad robotics\n");
           break;
   }
   InitSystem();
   WakePorts();
   MainLoop();
}
```

Notes:

o Very low overhead. Recommended for ROM applications where real estate is at a premium but some of the functions of printf() are required.

o The symbol error() is defined in STDIO.H. The word error is not a library global symbol.

o Not to be confused with ferror or perror.

The exit Function

Purpose:

Standard exit. Flushes buffers and closes the file control block to the disk directory for stream stdout.

Function Header:

```
VOID exit()
```

Returns:

Nothing. Exits to warm boot.

Example: Use of exit().

```
#include <stdio.h>
main()
{
     printf("Standard C exit\n");
     exit();
}
```

Notes:

- Calls the routine Cexit_(), then falls into the abort exit.

- If the Cexit_() routine is the one provided, then buffer flushing and closing will occur for at least stream stdout. But any other open streams will be ignored. Open files will leave orphan disk directory entries with 0 records.

- To change the character of the standard exit requires a new routine Cexit_(). Most systems flush all buffers and close all file control blocks back to the disk directory. See Cexit_() for an example of how to re-write it to simulate such a system.

The exp Function

Purpose:

Compute the exponential of real float value x

Function Header:

```
float exp(x)        Power with base e = 2.718 and exponent x.
float x;            Positive, negative or zero argument.
```

Returns:

number	x in range
INF	x too large positive (see notes).
0	x too large negative (see notes).
	errno = ERANGE returned to flag error.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float exp();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("exp(%f) = %f, errno = %d\n",x,exp(x),errno);
}
```

Notes:

- o Method used is exp(x) = pow(EBASE,x) where EBASE = 2.71828 = e = base for natural logarithm, approximately.
- o The error range is found by solving exp(x) = 1.0e39.

The expand Function

==

Purpose:

Expands wild card file names on the command line into a complete list. The list is passed to main() by the usual argc, argv[] method.

Function Header:

```
expand(a,b)
int *a;                 Address of argc variable
                        Usually you insert &argc
char *b;                Address of array argv[]
                        Usually you use &argv
```

Returns:

New argc value, old value is lost.
New table *argv[] with expanded wild cards.
Duplicates from expansion are ignored.

Example: Normal wildcard expansion in a C program. This example displays the wildcard expansions.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
 int i;
   expand(&argc,&argv);
   for(i=0;i<argc;++i)
     printf("argv[%d] = %s\n",i,argv[i]);
}
```

Example: Wildcard expansion of a command line inside a program. Commas or blanks may separate names. Note &-operator usage and the extra pointer char **bn.

```
#include <unix.h>
#include <stdio.h>
main()
{
 int i;
 int a;
 char buf[129];
 char *b[128];
 char *p,*name,**bn;
 char *sob(),*fnb();

    b[0] = "Unused"; a = 1; p = name = buf;
    printf("Enter wildcard names: "); gets(buf);
    while(*p) {if(*p == ',') *p = ' '; ++p;}
    do {
        name = sob(name);
        if(name[0]) b[a++] = name;
        p = fnb(name); if(p[0] == '\0') break;
        p[0] = '\0'; name = p+1;
    } while(*name);
    bn = b;
    expand(&a,&bn);
    for(i=0;i<a;++i)
        printf("b[%d] = %s\n",i,bn[i]);
}
```

Example: Break a command line into tokens using the tokens() routine in the library. No wildcard expansion.

```
#include <stdio.h>
main()
{
 int i,a;
 char buf[129];
 char *b[100];

    printf("Enter a C-style command line\n: ");
    gets(buf); if(buf[0] == '\0') exit();
    b[0] = b[1] = "Not in use";
    a = tokens(b,buf);
    printf("Re-direction files: %s, %s\n",b[0],b[1]);
    for(i=0;i<a;++i)
        printf("b[%d] = %s\n",i+2,b[i+2]);
}
```

Notes:

- o It can run out of room. Uses sbrk() to get the space for the expansions.

- o Takes 600 bytes of stack space.

- o CP/M-80 lets you access the command line from the default buffer. Copy it to safety with: strncpy(buffer,129,peekb(128));

- o The above won't work with re-direction because C/80 uses the default buffer to open the < & > files before running main.

- o See also TOKENS.C and the tokens() routine for a method of decoding a C command line.

- o Complete source code for expand(). It is often the case that you need a similar expansion function, with minor changes. Here is food for thought:

```
static int nomatch(n,f,t)
int n; char *f[],*t;
{
  while(--n) {if(strcmp(f[n],t)==0) return 0;} return 1;
}
#define MAXFILES        255    /* max number of expansions */
expand(argcp,argvp)
int *argcp; char **argvp;
{
 char *fn[MAXFILES];
 static int j,nf,c;
 static char **v,*arg,*p;
 char *core(),*decodF(),*strcpy(),*index();
 char *findfirst(),*findnext();

  c = *argcp; v = *argvp; fn[0] = v[0];
  for (nf=1,j=1;((nf < MAXFILES) && (j < c)); ++j) {
    cvupper(arg=v[j]);
    if(index(arg,'?') || index(arg,'*')) {
      p = findfirst('\0',arg);
      while (p != (char *)(-1)) {
        p = decodF(p);
        if(nomatch(nf,fn,p))
          strcpy(fn[nf++]=core(1+strlen(p)),p);
        p = findnext();
      }
    }
    else if(nomatch(nf,fn,arg)) fn[nf++] = arg;
  }
  *argcp = nf; fn[nf++] = -1;
  v = *argvp = core(sizeof(char *) * nf);
  while(nf--) v[nf] = fn[nf];
}
```

The fabs Function

Purpose:

Compute the absolute value of a float number.

Function Header:

```
float fabs(value)             Float Absolute value
float value;                  Float argument, 32 bits
```

Returns:

The absolute float value of the argument.

Example: Print the float absolute value of a floating point number entered at the console.

```
#define pfFLOAT 1
#include <stdio.h>
main()
{
char s[129];
float f,fabs(),atof();

     printf("Enter float number f: ");
     gets(s); f = atof(s);
     printf("fabs(%f) = %f\n",f,fabs(f));
}
```

Notes:

o Not a macro. But other libraries will use one. Watch out when you write your code, and when you port it to other machines.

o There are no side effects. This is an honest function.

o The cast must appear explicitly in your program. For example, if you use fabs(), then include a declaration of the form

```
float fabs();
```

o To print out floats using printf(), employ the header file switch called pfFLOAT. See the example above.

o A common error is to write g = abs(f) where f,g are floats. This has undefined results with no compile-time error reporting.

The fclose Function

==

Purpose:

Flush the stream buffer to disk. Close the file control block to the disk directory. Remove the stream pointer from system tables.

Function Header:

```
int fclose(fp)
FILE *fp;           Open stream pointer.
```

Returns:

EOF	On error.
0	Success.

Example: Close the stream stdout.

```
/* Command line is A>main >foo.bar */
#include <stdio.h>
main()
{
    printf("This line is sent to stream stdout\n");
    fclose(stdout);
}
```

Notes:

o An fclose() operation will fail if the buffer flush does not complete due to a disk error or lack of disk space. It can also fail due to a write-protect tab.

o An fclose() on a device sends an end of file character in case the device requires one. This is true for the printer LST:. It gets ctrl-L.

o An fclose() on the console is a NOP.

o Streams with descriptors 252,253,254,255 are devices. They are treated differently than files during I/O because there is no buffering.

o Programs written with re-direction in force always call Cexit_() to close stdout. The close is done by fclose().

The fdopen Function

==

Purpose:

Open a stream file using an existing file descriptor. Applies to disk files only. No devices allowed.

Function Header:

```
FILE *fdopen(fd,mode)
int fd;                     Open file descriptor
char *mode;                 Mode from fopen().
```

Returns:

0	Open failed
fp	Open worked, fp=stream pointer

Example: Create a TMP file by descriptor, then re-open as a stream.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int fd;
FILE *fp,*fopen();
     fd = creat("TMP",0);     /* protection mode ignored */
     if(fd != -1) {
       fp = fdopen(fd,"w");
       fputs("It worked\n",fp);
       fclose(fp);
       puts("TMP created");
     }
     else puts("Open failure on TMP");
}
```

Notes:

o In this library, all files that use file descriptor tables are also buffered files. Do not expect this feature on other systems.

o The function fdopen() can change the access mode of the file.

o Fdopen() uses a call to freopa(). Since the latter can handle ascii and binary mode changes, no further code is needed.

The feof Function

Purpose:

Tests for end of file on a stream.

Function Header:

```
int feof(fp)
FILE *fp;           Open stream pointer.
```

Returns:

0	Not at end of file
nonzero	Stream at end of file

Example: Check for end of file when using getw().

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
     while(TRUE) {
        if((x = getw(stdin)) == EOF) if(feof(stdin)) break;
        putw(x,stdout);
     }
}
```

Notes:

- o Testing putc() is impossible, since C/80 normally aborts when it runs out of disk space.
- o Files opened for write are at end of file, unless seek() has been used, in which case the use of feof() makes no sense at all.
- o The best way to understand feof() is to study its source code:

```
int feof(fp)
FILE *fp;
{
static int x,fd;
extern char IOmode[];
   fd = fileno(fp);
   if(IOmode[fd]=='w' || (x = getcbinary(fp)) == EOF ||
      (x == 26)) return 1;
   ungetc(x,fp);
   return 0;
}
```

The ferror Function

Purpose:

Tests for a stream error. This function returns 0 under C/80.

Function Header:

```
int ferror(fp)
FILE *fp;          Open stream pointer.
```

Returns:

0	Always.

Example: Compile Unix code under C/80.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int x;
     while((x = getchar()) != EOF) {
          if(ferror(stdout)) break;
          putchar(x);
     }
}
```

Notes:

- o ferror() is a no-op under C/80 because no provision has been made to survive a disk output error.

The fflush Function

==

Purpose:

Writes all buffered data onto the disk without changing file pointers or disturbing character I/O functions. Writes the current file control block into the disk directory, but leaves the file open for I/O.

Function Header:

```
#include <unix.h>          Service is #define fflush uflush
int fflush(fp)
FILE *fp;                  Stream pointer for a disk file
```

WARNING: The header file UNIX.H defines fflush as uflush. The main library has a function called fflush(), which does the same flush operation, but fails to update the disk directory or check for devices. If you want to use both, then **#undef fflush** and call each by its internal library name, fflush or uflush.

Returns:

```
 0       Success
-1       Failure
```

Example: Flush random file block to disk during I/O wait.

```
while(TRUE) {
  if(bdos(11,0)) return getchar();
  if(dirtyblock) { fflush(fp); dirtyblock=0;}
}
```

Notes:

- o The uflush() function writes the disk buffer to disk and then the file control block to disk.
- o The main library fflush() does not check for devices. It does not write the file control block back into the disk directory. The symbol FFLUSH is retained for old software compatibility.
- o New software should use UNIX.H. Direct reference to uflush() should be avoided.
- o Under C/PM 2.2, devices like the console and printer are not files, so uflush() does not act on these devices. Devices do not need flushing under CP/M because they are not buffered. When you print to the console or printer, it happens a byte at a time.

The fgets Function

==

Purpose:

Reads up to n characters from an open stream and null-terminates the string. Breaks before n characters if end of file or a newline is encountered.

Function Header:

```
char *fgets(s,n,fp)
char *s;                    Base address of the storage area.
int n;                      Maximum size of storage area in bytes.
FILE *fp;                   Open stream pointer.
```

Returns:

(char *)0	End of file and string empty.
(char *)0	Stream error and partially filled string.
(char *)s	Success, base of storage area.

Example: Echo lines typed at the console until ctrl-Z.

```
#include <stdio.h>
main()
{
char s[129];
     while(TRUE) {
          fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
          if(fgets(s,40,stdin) == (char *)0) break;
          fprintf(stderr,"%s",s);
     }
}
```

Notes:

- It is impossible to tell whether a null pointer return stands for an error or end of file.

- The array s[] may be filled with invalid data in case of an error.

- Newlines read from the stream are appended to the array s[]. This is different from gets(), which strips the newline on input. In practise, it means puts() is not useful for input obtained from fgets().

The fileno Function

==

Purpose:

Returns the file descriptor for a stream.

Function Header:

```
int fileno(fp)
FILE *fp;           Open stream pointer.
```

Returns:

```
fd        File descriptor for stream fp.
```

In the C/80 implementation, fd is the device handle 252, 253, 254 or 255, or fd is in the range 1..MAXCHN-1, which represents the array index which accesses the file information in IOTABLE.

Example: Use read() on an open stream fp.

```
#include <unix.h>
#include <stdio.h>
main()
{
char *buf,buffer[128];
FILE *fp,*fopen();
int fd,n;

  fp = fopen("MYFILE","r");
  if(fp) {
    buf = buffer;
    fd = fileno(fp);
    n = read(fd,buf,128);
    while(n--) putchar(*buf++);
    fclose(fp);
  }
}
```

Notes:

o Use fileno() to maintain Unix System III compatibility.

o While C/80 treats streams and descriptors as the same object, other systems do not.

o Read() and write() use descriptors rather than streams. This area of distinction accounts for most compile-time errors for older C/80 source code using Unix-style compilers.

The fillchr Function

==

Purpose:

Fills a region of memory with a byte value.

Function Header:

```
char *fillchr(dest,c,n)
char *dest;                   Destination address.
char c;                       Byte value for fill.
int n;                        Number of bytes to fill.
```

Returns:

```
dest+n                        Next location after fill.
```

Example: Fill a buffer with nulls.

```
#include <stdio.h>
main()
{
char *p,*fillchr();
char s[100];
     p = fillchr(s,'\0',100);
     printf("s = %u, fillchr(s,'\0',100) = %u\n",s,p);
}
```

Notes:

- o Some libraries call this function setmem(). The ordering of the arguments is different, however. We use the same ordering as is used by strncpy() and strncat(). It looks like setmem() has assembly language origins for an Intel CPU. It is not a Bell Labs Unix V7 function, so let the argument continue.
- o The integer n could be unsigned, because this function does its arithmetic using unsigned compares. For portability we recommend restraining its argument to type integer.
- o PASCAL has a function called fillchar() that is similar to the one implemented in this library. It accounted for glowing claims about the speed of PASCAL over C when the first BYTE Sieve of Eratosthenes Benchmark was published (Sept, 1981).

The fin and fout Variables

Purpose:

File descriptors fin and fout for streams stdin and stdout.

Example: Display the file descriptors for the standard input and output streams. Unix-compatible calling sequence.

```
#include <unix.h>
#include <stdio.h>
main()
{
  printf("fin = %d, fout = %d\n",fileno(stdin),fileno(stdout));
}
```

Example: Display the file descriptors for the standard input and output streams. Direct method.

```
#include <stdio.h>
main()
{
extern int fin,fout;
  printf("fin = %d, fout = %d\n",fin,fout);
}
```

Notes:

- This library simulates stream pointers in STDIO.H by means of special functions fin__() and fout__().
- The function fileno() returns a file descriptor for a given open stream pointer.
- Descriptors fin and fout are both 0, which refers to the console device, unless re-direction is in force.
- The naming conventions for the descriptors comes from obsolete versions of Unix. Beware in using fin and fout when trying to write portable code.

The fin__ and fout__ Functions

Purpose:

Return the file descriptors for streams stdin and stdout.

Function Header:

```
int fin__()
int fout__()
```

Returns:

```
fin__()        The file descriptor for stream stdin
fout__()       The file descriptor for stream stdout
```

Example: Define macros for streams stdin and stdout.

```
#define FILE int
#define stdin (FILE *)fin__()
#define stdout (FILE *)fout__()
```

Notes:

- o The above macros are implemented in STDIO.H.
- o Meaningless statements like stdin = 0 will hopefully produce compiler error messages.
- o The fact that fin and fout are extern variables will cause compile errors, unless they are declared properly. You can usually use fin__() and fout__(), thereby avoiding the extern declarations.
- o The functions fin__() and fout__() present a portable interface across the versions of Unix. It is best not to use these functions at all. However, if you must, then it is better to bury the problems in fin__() and fout__() than to directly use the externs fin and fout.

The findFIRST Function

==

Purpose:

Finds a disk file by name or wildcard file spec.

Function Header:

```
char *findFIRST(extent,filename)
char extent;                            Extent, '\0' is normal.
char *filename;                         Name of file, null-terminated.
```

Returns:

```
(char *)-1          Failure
(char *)p           p = address of matching file control block
                    taken from the disk directory. Actually, p
                    points into the default buffer at 0x80.
```

Example: Search for a wildcard file name and print the first one found.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
char *p;
char *decodF(),*findFIRST();
     if(argc <= 1) exit();
     if((p = findFIRST('\0',argv[1])) != -1)
          printf("First match = %s\n",decodF(p));
     else printf("File not found\n");
}
```

Notes:

- o Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.
- o An extent of '?' will search all user areas.
- o A file name of "*.*" will search for all files. The question mark wildcard also works.
- o This function uses the default buffer at BOOT+0x80. In most applications you have to copy the file control block to safety or else use decodF() and copy the return string to safety.
- o The findFIRST() function can be simulated on other systems by using makeFCB(fcb,file) and bdos(17,fcb). The latter returns the directory entry offset (mod 32) into the default buffer.

The findNEXT Function

Purpose:

Finds the next file name match after a call to findFIRST().

Function Header:

```
char *findNEXT()
```

Returns:

(char *)-1	Failure
(char *)p	p = address of matching file control block taken from the disk directory. Actually, p points into the default buffer.

Example: Search for a wildcard file name and print all occurrences.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
int i;
char *p;
char *decodF(),*findFIRST(),*findNEXT();
     if(argc <= 1) exit();
     i = 0;
     p = findFIRST('\0',argv[1]);
     while(p != (char *)-1) {
       printf("%s\n",decodF(p));
       p = findNEXT();
       ++i;
     }
     if(i == 0) printf("File not found\n");
}
```

Notes:

o Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.

o This function uses the default buffer at BOOT+0x80. In most applications you have to copy the file control block to safety or else use decodF() and copy the return string to safety.

o findNEXT() MUST follow an initial call to findFIRST() without any intervening disk I/O than can disrupt the default buffer.

o On other systems, findnext() is just bdos(18,0) or ccBDOS(18).

The fixfile Function

==

Purpose:

Inserts drive name, file name and extension defaults into a string that contains a supposed file name. Does not process wildcards.

Function Header:

```
     VOID fixfile(filename,pattern)
     char *pattern;                     Pattern to be used to fix the
                                        file name.
     char *filename;                    Name of file, null-terminated.
                                        Must be large enough to receive
                                        the fix.
```

Returns:

Nothing. Area assigned to the filename is over-written.

Example: Fix a file name entered by the user.

```
     #include <stdio.h>
     main()
     {
     char *p,s[129],t[129];
          printf("Enter a file name: ");
          gets(t);
          strcpy(s,t);
          fixfile(s,p="A:");
          printf("pattern = %s, fixed file name = %s\n",p,s);
          strcpy(s,t);
          fixfile(s,p="B:.C");
          printf("pattern = %s, fixed file name = %s\n",p,s);
          strcpy(s,t);
          fixfile(s,p="A:MYFILE.TXT");
          printf("pattern = %s, fixed file name = %s\n",p,s);
     }
```

Notes:

o A null file name will cause the pattern to be used as the complete file name - the pattern is the default file name.

o If the pattern contains a drive spec and the user input does not then the file name is fixed: the drive spec is inserted. A similar action is taken for the other two fields: file name and extension. If the pattern contains a blank field, then no fix is done to the user file name.

o To use fixfile to add a default extension of .COM, use the pattern ".COM". In this case, the user is forced to add the drive name and the file name. The extension .COM will be appended by fixfile if the user entered no extension.

The floor Function

==

Purpose:

Compute float floor, the largest integer less than or equal to the given number. For example, floor(-1.1) = -2 and floor(1.1) = 1.

Function Header:

```
float floor(f)
float f;              Float argument.
```

Returns:

```
g                Where g is a whole number, signed,
                 with g <= f and f < g+1.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float f;
float atof();
float floor();
     printf("Enter float f: "); gets(s); f = atof(s);
     printf("floor(%f) = %f\n",f,floor(f));
}
```

Notes:

o No error codes returned. No need to code for errno.

o Floor() does not truncate. For x > 0, truncate, but for x < 0 subtract one and truncate.

o No overflow check.

o No auto conversion to FLOAT.

The fmod Function

===

Purpose:

Solves for float f in the equation x = k*y + f such that x*f >= 0, fabs(f) < fabs(y), for some long integer k.

Function Header:

```
float fmod(x,y)
float x,y;          Arguments x, y as above.
```

Returns:

f — As computed above, a remainder (not a modulus).
For this float library, k = x/y and f = x - k*y.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,y;
float atof();
float fmod();
     printf("Enter float x: "); gets(s); x = atof(s);
     printf("Enter float y: "); gets(s); y = atof(s);
     printf("fmod(%f,%f) = %f\n",x,y,fmod(x,y));
}
```

Notes:

- o No error codes returned. No need to code for errno.
- o No overflow check.
- o No auto conversion to FLOAT.
- o Fmod() works because the float library converts to integer by truncation. If it didn't, then it would be much more difficult to write.

The fnb Function

==

Purpose:

Skips over non-blank characters to locate the first blank. Stands for Find Next Blank. Blanks are defined by isspace().

Function Header:

```
char *fnb(s)
char *s;                  String, null-terminated.
```

Returns:

The address of the null terminator in the string, or the address of the first blank character. Blanks are defined by isspace().

Example: Find and print the first word in a line of text.

```
#include <stdio.h>
main()
{
char buf[129];
char *s,*t,*sob(),*fnb();

     printf("Enter a line of text: ");
     gets(buf); s = sob(buf); t = fnb(s);
     while(s<t) putchar(*s++);
     putchar('\n');
}
```

Notes:

- This function is almost always used in conjunction with sob(), in order to isolate tokens in a line of text.

- High-overhead functions like scanf() can often be avoided in a portable way by using sob() and fnb().

- Both sob() and fnb() return the address of the null delimiter of the string in case the character class checking runs off the end of the string.

- See also the STRINGS PACKAGE for useful string utilities, all in portable C.

- The source code for fnb():

```
char *fnb(s) char *s;
{
  while(*s != EOS && isspace(*s) == FALSE) ++s; return s;
}
```

fopen_ - C/80 Standard fopen

The fopen_ Function for C/80

Purpose:

Opens a device or buffered file for I/O as a stream using the mode conventions in the Software Toolworks C/80 library.

Function Header:

```
#include <stdio.h>
FILE *fopen_(file,mode)
char *file;              Null-terminated file name.
char *mode;              Mode string. Options:
                         "r"      Read only. Must exist.
                         "w"      Write (or read/write/seek)
                         "u"      Update. Must exist.
                         "rb"     Binary "r"
                         "wb"     Binary "w"
                         "ub"     Binary "u"
```

WARNING: Use UNIX.H to get modes "r+","w+","a","a+". The above function is low-level. It matches usage in C/80 programs that use the Software Toolworks fopen().

Old C/80 code that uses fopen() should compile and link without change. New source code should use the symbol **fopen_.**

Returns:

```
(FILE *)0      Failure.
(FILE *)fp     File pointer on success.
```

Example: Open a file for read only to see if it exists.

```
#include <stdio.h>
main()
{
FILE *fp,*fopen_();

     if(fp = fopen ("TMP","r"))
     puts("File TMP exists");
     else
     puts("File TMP does not exist");
}
```

Notes:

o An fopen () operation will fail if IOch[] does not have an empty slot. There are 4 devices LST:,CON:,RDR:,PUN: plus MAXCHN-1 files (usually 6). The macro **nfiles** can be set to less than **MAXCHN-1**, but the number of slots is fixed.

o Devices do not require an fopen (). If you know the file descriptor, then simply read/write to that stream. Streams with descriptors 252,253,254,255 are devices.

o If nfiles = 3 and MAXCHN = 7 and you try to open file #4, then fopen () has to call sbrk() to get space for the file control block and the file buffer. This can fail. It can also succeed and fragment the heap, because the sfree() function won't free up a file buffer.

o Under Bell Labs Unix, a file descriptor fd is related to the stream fp by fd = fileno(fp).

o The C/80 fopen () is more limited than the general fopena() and fopenb() functions, which also allow append. Use UNIX.H to access the latter.

o To use more than 6 files, you have to re-compile IOTABLE.CC and put the new IOTABLE.REL into the library or else link in IOTABLE.REL as a separate module.

o In the C/80 library, read() and write() communicate with files only and operate on quanta of 128 bytes. If you use UNIX.H, then all the rules change. See read() and write() for details plus UNIX.H.

o For safety, use only getc() and putc() when operating on devices CON:, LST:, RDR:, PUN:. The resulting code is much more portable and substantially easier to debug.

o A call to fopen () in mode "w" creates a new file if it does not exist. If it does exist, then it is erased from the disk directory and a new file is created.

o A call to fopen () in mode "u" opens the file in "r" mode and then changes its flag to "u". So files opened for update must already exist. Generally, this is the mode used for random files using seek(). Such files are read/write.

The fopen, fopena Functions

===

Purpose:

Unix Ascii File fopen() implemented for C/80.

Function Header:

```
FILE *fopena(name,mode)
char *name;                   Ascii file name, null-terminated
char *mode;                   Mode string, see below
```

Returns:

```
(FILE *)0           Open failed
(FILE *)fp          Open worked, fp=stream pointer
```

Example: Open a file for append, creating it as necessary, ready to write on the end.

```
#include <unix.h>
#include <stdio.h>
main()
{
FILE *fp,*fopen():
  fp = fopen("MYFILE","a+");
  if(fp) {
    fputs("This goes on the end\n",fp);
    fclose(fp);
  }
}
```

WARNING: In UNIX.H we make this definition:

```
#define fopen  fopena
```

Both fopena(), fopenb() are honest functions. See FOPENB.C for details about fopenb().

Notes:

- Use of this function requires seekend() which adds about 300 bytes to the object code size.

- Unix does not support binary modes. The Unix Modes are r,w,a,r+,w+,a+ as follows:

  r open for read, file must exist

  w open for write, any existing file with the same name is truncated

  a open for append, add onto end of an existing file or create a new one.

  r+ open for reading and writing starting at the beginning of the file. The file must already exist.

  w+ open for reading and writing starting at the beginning of the file. The file is truncated if it already exists, or created if it does not.

  a+ open for reading and writing starting at the end of the file. The file is created if it does not already exist.

- For maximum portability, use stub functions fopena and fopenb for Ascii and Binary opens. These functions present a portable interface to most libraries.

- Binary modes are supported by both fopen() and fopena() library functions. However, the mode string method will cause grave portability problems. The following mode access strings will work. However, it is recommended that you use fopenb() to achieve the desired result.

  rb Binary mode for r (existing file, read binary)

  rb+ Binary mode for r+ (existing file, update binary)

  wb Binary mode for w (write binary)

  wb+ Binary mode for w+ (overwrite binary update)

  ab Binary mode for a (create or binary append)

  ab+ Binary mode for a+ (create or append, binary update)

  u Same as r+

  ub Same as rb+

The fopenb Function

==

Purpose:

Binary File fopen() for C/80. Not a Unix standard function.

Function Header:

```
FILE *fopenb(name,mode)
char *name;                        Null-terminated file name
char *mode;                        Mode string, see fopen().
                                   Usually "r", "w", "a".
```

Returns:

0	Open failed
fp	Open worked, fp=stream pointer

Example: Open a binary file for read, count all the carriage returns.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x,n;
FILE *fp,*fopenb();
  fp = fopenb("MYFILE","r");
  n = 0;
  if(fp) {
    while((x = getc(fp)) != EOF) {
      if(x == '\r') ++n;
      putchar(x);
    }
    fclose(fp);
    printf("\n%d carriage returns\n",n);
  }
}
```

WARNING: In UNIX.H the following definition is made:

```
#define fopen   fopena
```

Note that fopena(), fopenb() are honest functions and not macro definitions.

Notes:

- o Use of this function requires seekend() and its corresponding overhead of about 300 bytes.

- o Unix does not support binary modes. The Unix Modes are r,w,a,r+,w+,a+. See fopena.c.

- o For maximum portability, use the functions fopena() and fopenb() for Ascii and Binary opens. These functions present a portable interface to the various modes.

- o Binary modes are supported by both fopen() and fopena() library functions. However, mode string usage differs across compiler libraries. To be safe, use fopenb().

The fprintf Function

Purpose:

Outputs formatted data to an open stream using a control string and an appropriate argument list of variable length.

Function Header:

```
fprintf(fp,control,arg1,arg2,...);
FILE *fp;                           open stream
char *control;                      control string, see below
arg1,arg2,...                       appropriate arguments,
                                    8, 16 or 32-bit data, see below
```

Returns:

Nothing.

Example: Print an Ascii chart in Decimal, Octal, Hex, Binary.

```
#include <stdio.h>
main()
{
int i;
  for(i=0;i<128;++i)
     fprintf(stdout,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
}
```

Notes:

- o Syntax and usage follows printf() exactly. See printf() for all the tricks.
- o In this library, fprintf() is supplied in two versions. The fast version lacks some of the features found in the denser version for longs and floats. Both versions support multiple arguments.
- o Fprintf() is not recursive. You cannot do something like fprintf(stdout,"%s",f(s)) where f(s) calls sprintf(). Across libraries, such coding appears to be non-portable.

The fputs Function

Purpose:

Prints a null-terminated string to an open stream. Writes all characters in the string, except the terminating null.

Function Header:

```
int fputs(s,fp)
char *s;                  Source string base address
FILE *fp;                 Open stream pointer
```

Returns:

0	Operation completely successful.
EOF	Error occurred.

Example:

```
#include <stdio.h>
main()
{
char s[129];
     fprintf(stderr,"Enter a string: ");
     gets(s);
     fputs(s,stderr);
     fputs("\n",stderr);
}
```

Notes:

o A newline is NOT appended, as is the case for puts().

o If the string is void, then no character is output and 0 is returned. If an error occurs or End of Media is detected on the output file, then EOF is returned.

o Unfortunately, the C/80 standard under CP/M is to bail out to the system when an output error occurs. This is because CP/M cannot gracefully handle a full disk situation.

o The standard for fputs() requires that some character other than EOF be returned if the operation succeeds. We chose 0, rather than the last character output by putc(), in order to avoid problems with sign extension.

The fread Function

==

Purpose:

Transfers bytes from a stream file to main memory. Move bytes in n packets of size s.

Function Header:

```
int  fread(buff,s,n,fp)
char *buff;                       Buffer of data
int  s;                           Size of each data element
int  n;                           Number of elements
FILE *fp;                         Stream pointer, open stream
```

Returns:

n	The number of packets of size s that were actually transferred.
0	Error, including end of file

Example: Read 10 long integers into an array.

```
#include <unix.h>
#include <stdio.h>
main()
{
auto
long nl[10];
FILE *fp,*fopenb():
  fp = fopenb("LONGDATA.DAT","r");
  if(fp) {
     fread(nl,sizeof(long),10,fp);
     fclose(fp);
  }
}
```

Notes:

- o Use feof() to distinguish a disk error from end of file.
- o Useful for reading in more than 64k without the use of a program loop.
- o The primitive read() is used implicitly, but all Unix redirection is in force, because of the explicit use of getc().
- o fread() is slower than the primitive read().
- o The UNIX.H definition of read() is reada(), which does slow, correct Unix I/O.

The free and cfree Functions

Purpose:

Free() releases memory assigned by malloc(). Cfree() is a synonym often used with calloc().

Function Header:

```
int  free(a)
char *a;                 Character pointer to contiguous area
                         obtained from malloc().

int  cfree(a)
char *a;                 Character pointer to contiguous area
                         obtained from calloc().
```

Returns:

```
 0   if the request to free the area was honored
-1   if the request failed
```

Example: Get 2048 bytes from malloc(), then free it.

```
#include <unix.h>
#include <stdio.h>
getmem()
{
char *p;

     p = malloc(2048);
     if(p == (char *)0) puts("Request failed");
     else
     free(p);
}
```

Structure used by malloc(): The following 4-byte header appears just before the base address returned by malloc(). It is used by both malloc() and free(), so be careful not to corrupt it.

```
struct block {
          struct block
              *nxtblk;
          unsigned
                  siz;
};
```

Notes:

o Code for free() derived from K&R(1978), pp 174-177.

o Brk() and Sfree() don't know about malloc() or free().

The freopen, freopa Functions

==

Purpose:

Substitutes a new file for an open stream. The name and access mode may be changed during the call.

Function Header:

```
FILE *freopen(name, mode, fp)
FILE *freopa(name, mode, fp)
char *name;                    File name, null-terminated
char *mode;                    Mode "r", "w", "a", see below
FILE *fp;                      Stream pointer, open stream
```

WARNING: In UNIX.H the following definition is made:

```
#define freopen freopa
```

The function freopa() is an Ascii re-open function. It is not supposed to implement binary I/O. The binary re-open function is called freopb().

Example: Re-assign stream stdout to a file at run time without using re-direction on the command line. This example appends a fixed output file each time it runs.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern int fout;
FILE *fp,*freopen():
  fp = freopen("OUTPUT","a",stdout);
  fout = fileno(fp);
  printf("This message should go to file OUTPUT\n");
}
```

Notes:

o Always closes the existing stream before the re-open. This function does not do binary opens. See freopb().

o Descriptors fin and fout are fixed as required.

o It so happens that this particular freopa() can handle 3-char mode strings and hence binary opens. However, for maximum portability, use freopb().

o There is no way to re-assign the descriptors fin and fout to reflect stdin and stdout changes without going through the process in the example. This is one of the very few places where the Unix V7 standard is violated.

The freopb Function

===

Purpose:

Substitutes a new file for an open stream. The name and access mode may be changed during the call. The new open is done in binary mode.

Function Header:

```
FILE *freopb(name, mode, fp)
char *name;                     File name, null-terminated
char *mode;                     Mode string "r", "w", "a"
FILE *fp;                       Stream pointer, open stream
```

WARNING: In UNIX.H the following definition is made:

```
#define freopa  freopen
```

The functions freopen and freopa are Ascii re-open functions. They cannot implement binary.

Example: Re-assign stream stdin from a command line re-direction argument. This example reopens stdin as a binary file so we can read characters without CR/LF translation. The example counts the number of CR/LF pairs.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern int fin;
extern char *p_fin;
int x,y;
FILE *freopb():
  if(p_fin == (char *)0) exit();
  fin = fileno(freopb(p_fin,"r",stdin));
  y = 0;
  while((x = getchar()) != EOF && x != 26) {
     if(x == '\r' && getchar() == '\n') ++y;
  }
  printf("File %s contains %u lines\n",p_fin,y);
}
```

The descriptor fin and the file name pointer p_fin are special to this library. It is here that we see some real differences between Bell Labs Unix V7 and the present upgrade from Unix V3. However, the use being made is not portable to Unix anyway - binary I/O is not featured.

Notes:

- o Always closes the existing stream before the re-open.
- o File descriptors fin and fout are fixed as required.
- o This function does binary opens. See freopen() for Ascii re-opens.
- o There is no way to re-assign the descriptors fin and fout to reflect stdin and stdout changes without going through the process in the example. This is one of the very few places where the Unix V7 standard is violated. See also freopen, freopa.

The frexp Function

Purpose:

Splits x into x = f*(radix**n) where n is an integer and the value of f satisfies 0.5 <= f < 1.0. The value of radix is 2 for this library.

Function Header:

```
float frexp(x,nptr)
float x;                  Float to be split into mantissa &
                          exponent.
int *nptr;                Where to store the integer exponent.
```

Returns:

f	The fraction, a FLOAT.
n	via *nptr = n;

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
int n;
float x,f;
extern int errno;
float atof();
float frexp();
  errno = 0;
  printf(
  "frexp(x,nptr) splits float x into mantissa & exponent\n");
  printf("Enter a float x: "); gets(s); x = atof(s);
  f = frexp(x,&n);
  printf("frexp(%f,nptr) = %f, *nptr = %d, errno = %d\n",
         x,f,n,errno);
}
```

Notes:

- o The float exponent in this library is r = ((int)255 & (int)x.c[3]) - (int)128, where the float is expressed as a union { char c[4]; float f; } x. The float library is radix 2.

- o There is always a better way to do this calculation based on knowledge of the float representation.

fscanf

The fscanf Function

==

Purpose:

Parses formatted input text from a specifed open stream.

Function Header:

```
#include <stdio.h>
int fscanf(fp,control,&arg1,&arg2,...)
FILE *fp;                                  Open stream pointer
char *control;                             Control string, see below
&arg1,&arg2,...                              Appropriate arguments
                                           See below for rules.
```

Returns:

The number of successfully parsed arguments. An error occurred if the number returned does not match the number of arguments following the control string. **C/80 requires that fscanf() be enclosed in parentheses** in order to check the return value, i.e.,

```
                  i = (fscanf(fp,"%d",&x));      RIGHT WAY
rather than
                  i = fscanf(fp,"%d",&x);        WRONG WAY
```

Example: The following reads numbers from a file entered on the command line until either a data file error occurs or end of file is reached.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
FILE *fp,*fopen();
int x;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((fscanf(fp,"%d",&x))>0) printf("%d\n",x);
}
```

Notes:

- o The scanf family of functions accepts only addresses for its argument list. To error-check coding, verify that each argument has the address operator & as a prefix, or else the argument is a pointer (hence already an address).

- o The converted values are stored at the given addresses in order left to right. Skips in the control string do not have an argument so the argument count may not match the conversion count.

- o Short counts may hang the run-time package. Overly abundant counts will leave variables unfilled at run time.

- o Always check the return of fscanf() to see if matches the expected value. It is easy to program infinite loops using fscanf().

- o C/80 and its multiple-argument kludge cause us to write parentheses around fscanf() in order to recover the returned value. For example,

```
                if((fscanf(fp,"%d",&x)) > 0)
```

  will not work under C/80 with the extra parentheses removed.

- o To turn on float or long libraries for use with fscanf(), use the appropriate switches:

```
#define sfLONG  1        /* turn on long fscanf */
#define sfFLOAT 1        /* turn on float fscanf */
```

  The compiled code will change in size according to how much of the float and long libraries are actually used.

The fseek Function

===

Purpose:

Positions the read/write pointer in an open file. Has no effect on devices like the console or printer.

Function Header:

```
int  fseek(fp,offset,position)
FILE *fp;                           Stream pointer, open stream
long
offset;                             Long integer offset from position
                                    described below. Offset can be
                                    negative.
int
position;                           0 = Beginning of file
                                    1 = Current position in file
                                    2 = End of the file
```

Returns:

0	It worked
-1	It failed

Example: Seek to the last record of a CP/M file that was opened in binary mode.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
int x;
FILE *fp,*fopenb();

  if(argc <= 1) exit():
  fp = fopenb(argv[1],"r");
  if(fp) {
     if(fseek(fp,-128L,2) == 0) {
       puts("seek worked");
       while((x = getc(fp)) != EOF) putchar(x);
     }
     else
     puts("Bad seek");
  }
}
```

WARNING: Numbers like 128 are 16 bits whereas 128L is 32 bits. Look out for stack misuse with functions like fseek.

Notes:

- o Requires the long/float library in order to be used. This has been automated through the header file STDIO.H. If it doesn't work, then use #define mathlib 1 to get the required library search.

- o The largest file possible under CP/M-80 is 65536*256 = 16777216 bytes.

- o The second argument of fseek() is a long, 32 bits. The most common usage error is to write in a small number not cast as a long integer. In the example we used -128L. To use -128 instead is a serious error, as it causes the stack to be off by 16 bits. Note that the error would not occur if we used a variable for the second argument or if the number used was larger than 32767.

The C/80 Standard ftell and ftellr Functions

Purpose:

Reports the position of the file pointer for an open stream.

Function Header:

```
int ftell(fp)
FILE *fp;                     Open stream pointer

int ftellr(fp)
FILE *fp;                     Open stream pointer
```

Returns:

ftell: Byte count from the beginning of the file, or for files over 64k bytes, the position of the file pointer in the current sector.

ftellr: Number of bytes from the beginning of the file divided by 256 (number of whole 256-byte sectors).

WARNING: The header file UNIX.H re-defined ftell to be ftellu, which returns a long integer. The above functions are primitives that are used by ftellu. For future compatibility, try to use the long integer return of ftellu (or ftell with UNIX.H).

Example: Print the file pointer position for an open stream.

```
VOID report(fp) FILE *fp;
{
int i; unsigned amt,ftell();

        amt = ftellr(fp);
        if(amt < 256) { /* then ftell gives byte count */
          printf("Pointer at byte %u\n",ftell(fp));
        }
        else {
          printf("Pointer at sector %u, byte %u\n",amt,ftell(fp));
        }
}
```

Notes:

- The Bell Labs Unix V7 compatible library CLIBU.REL uses ftellu() which returns a long integer answer. If you do NOT include the header file UNIX.H, then the standard C/80 ftell will be used.
- If you use UNIX.H, then ftell, ftellr and ftellu will appear in the symbol table from M80/L80. Your code will reference ftell() but actually use ftellu().

The ftell and ftellu Functions

Purpose:

Reports the position of the read/write pointer in an open disk file. Not for devices like the console or printer.

Function Header:

```
#include <unix.h>
long ftell(fp)
FILE *fp;                Stream pointer, open stream
```

Returns:

```
-1       Failure
lp       long integer offset of read/write pointer
         from the beginning of the file.
```

WARNING: In UNIX.H is made the definition

```
#define ftell    ftellu
```

This definition allows you to use the Unix calls with standard long integer arguments. If you want to mix and match, then undo the definition and use ftellu().

Example: Report the position in an open file.

```
#include <unix.h>
#include <stdio.h>
findPosition(fp)
FILE *fp;
{
long ftell();

     printf("Position of file is %lu\n",ftell(fp));
}
```

Notes:

- o Requires the long/float library in order to be used. This is automated in STDIO.H

- o The largest file possible under CP/M-80 is 65536*256 = 16777216 bytes.

- o You must make a declaration for ftell() as in the example. The header files and the library cannot do it for you.

- o It is a disaster to use ftell() returns as an integer. The return conventions for integers versus long are assumed to be different. It is an accident if it works at all.

The ftoa Function

==

Purpose:

Converts a 32-bit internal format floating point number into a null-terminated string of decimal digits. Decimal point, precision and exponent field options exist. SPECIAL C/80 MATHPACK FUNCTION.

Function Header:

```
ftoa(how,pr,f,s)
char how;                  Conversion 'E', 'F', 'e', 'f', 'g'
int pr;                    Precision, 6 is K&R default.
float f;                   Float to be converted to ASCII.
char *s;                   Storage string for digits and exponent.
```

Returns:

Storage s is filled with converted digits and exponent field, null-terminated. This area must be large enough to hold the conversion.

Example: Get a float from the console and print without using the printf() family of functions.

```
#include <stdio.h>
main()
{
float f,atof();
char s[129];

     fputs("Enter a float: ",stderr);
     gets(s);                          /* e.g., s="23e-1" */
     f = atof(s);                      /* f is 4 bytes */
     ftoa('f',6,f,s);                  /* e.g., s="2.300000" */
     fputs(s,stderr);
}
```

Notes:

o Floats are 32 bits (4 bytes). Storage of a float is documented in the C/80 Mathpack. See also the Transcendental function library for information about the floating point exponent.

o This function differs for every float library implementation. The best way to avoid facing the rules is to use sprintf().

o The source for ftoa() is provided in assembler with The Software Toolworks MathPack.

The fwrite Function

==

Purpose:

Transfers bytes from main memory to a stream file. Move bytes in n packets of size s.

Function Header:

```
int  fwrite(buff,s,n,fp)
char *buff;                     Buffer for data
int  s;                         Size of each data item
int  n;                         Number of data items
FILE *fp;                       Stream pointer, open stream
```

Returns:

```
n       The number of packets of size s that
        were actually transferred.
0       Error, including end of file
```

Example: Write a buffer of 4096 bytes to an output file. Return number of bytes actually written.

```
#include <unix.h>
#include <stdio.h>
int doWrite(fp,buf)
FILE *fp;
char buf[4096];
{
int x;
     x = fwrite(buf,1,4096,fp);
     return (x);
}
```

WARNING: Some libraries on other target systems foolishly write this function to some unknown standard that returns less than the number of packets requested, even though the write did not fail. They break on terminators like carriage return and line-feed. Beware as you try to transport this particular function.

Notes:

- o Useful for writing out more than 32k without the use of a program loop.
- o The primitive writea() is used implicitly, but all Unix redirection is in force, because of the explicit use of putc().

The getatt Function

Purpose:

Get the CP/M file attributes, which are:

	ATTRIBUTE NAME	SYMBOLS USED
1.	Read-only	$R/O or $R/W
2.	System	$SYS or $DIR
3.	Archive	Used by some backup programs, but not by CP/M at present

Function Header:

```
int  getatt(name)        CP/M coded attribute, see below
char *name;              File name, usual CP/M conventions
```

Returns:

```
-1       File not found, otherwise
 n       Coded attribute word, 0 <= n <= 7,
         where:

           n&1 = Read-only attribute
           n&2 = System attribute
           n&4 = Archive attribute
```

Example: Print the attributes of a CP/M file entered on the command line.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int n;
    if(argc <= 1) {
        puts("Usage: A>main filename"); exit();
    }
    if((n = getatt(argv[1])) == -1) {
        puts("File not found"); exit();
    }
    printf("Attributes for file %s\n",argv[1]);
    puts( (n&)1 ? "$R/O" : "$R/W");
    puts( (n&2) ? "$SYS" : "$DIR");
    puts( (n&4) ? "Archive set" : "Archive not set");
}
```

Notes:

- o The name is a normal C string with NULL delimiter.
- o To decode the attributes on return, use the logical AND operator & as outlined in the example above.

The getbyte Function

==

Purpose:

Reads a byte from the console, no-echo, even though going through the standard C library function getchar() to obtain the character.

Function Header:

```
int getbyte()
```

Returns:

x	The byte read from the keyboard, no translation, or from open stream stdin, in which case translation can occur.

Example: Get a yes or no answer from the user.

```
#include <stdio.h>
main()
{
     while(TRUE) {
       fputs("Do you want to continue? ",stderr);
       if(toupper(getbyte()) != 'Y') break;
       puts("YES");
     }
     puts("NO");
}
```

Notes:

o Below is the source code for getbyte(). It does a direct BIOS call via getchar() to obtain the character without console echo and without any kind of translation. The reason for the BIOS call is to allow NULL to be read as a character (CP/M function 6 disallows NULL).

```
int getbyte()
{
extern char Cmode;
int x,y;
  y = Cmode; Cmode = 2;
  x = getchar();
  Cmode = y;
  return x;
}
```

o The function getbyte() gets characters from files under stdin re-direction. The console I/O will be the same, but character translation will depend upon the open mode of the file. See freopen() for ways to invoke binary mode. See the Binary Flags section and GC_BIN for other ideas.

The getc Function

Purpose:

Reads the next character from an open stream. Not a macro.

Function Header:

```
int getc(fp)
FILE *fp;                   Open stream pointer
```

Returns:

-1	If end of file was reached. We define EOF to be -1.
c	Next character from stream fp, on success.

Example: The following reads characters from a file entered on the command line until either a data file error occurs or end of file is reached. Each character is printed as it is read.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int x;
FILE *fp,*fopen();
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((x = getc(fp)) != EOF) {
          printf("%c",x);
     }
}
```

Notes:

- o Expect getc to be a macro in most C libraries. It's not in this one.
- o Under CP/M, the BIOS will complain on a disk read error, which is the only possibility besides end of file. In this case, the system will kill the running program.
- o A return of -1 under C/80 means that end of file was encountered. Other C systems will require that you check EOF with feof(). The RDR: device never returns EOF, but CON: does in line mode.
- o Beware of mixing file descriptors and stream pointers. While C/80 will buy it, other systems won't. The connection is fd = fileno(fp), where fd is an integer and fp is a stream.
- o Both CON: and RDR: devices are recognized. The RDR: device does not translate characters, but CON: does unless in binary mode.

The getcbinary Function

Purpose:

Reads the next character from an open stream in binary mode. Not a macro. Recognizes CON: and RDR: devices. Not a K&R standard function.

Function Header:

```
int getcbinary(fp)
FILE *fp;                Open stream pointer
```

Returns:

-1	If end of file was reached. We define EOF to be -1. Disabled for CON: and RDR:.
c	Next character from stream fp, on success, with no translation of any kind.

Example: The following reads characters from stdin in binary mode until EOF or ctrl-Z is encountered. The total carriage return count is reported.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x,y;
     y = 0;
     while((x = getcbinary(stdin)) != EOF && x != 26) {
          if(x == '\r') ++y;
     }
     printf("%d returns\n",y);
}
```

Notes:

o Both CON: and RDR: devices are recognized.

o A return of -1 under C/80 means that end of file was encountered. The RDR: device never returns EOF, but CON: does in line mode. In binary mode, CON: returns the characters entered without translation.

o For portability, do not use this function. It is a library internal documented for your convenience. It is not supported on most systems.

The getchar Function

Purpose:

Reads the next character from the standard input stream stdin. Not a macro in this library.

Function Header:

```
int getchar()
```

Returns:

c	Next character from stream stdin, on success.
EOF	If end of file was reached. We define EOF to be -1. The end of file character for the console is ctrl-Z. For files (under re-direction), it is ctrl-Z or EOF.

Example: The following reads characters from the console until end of file is reached. Each character is printed as it is read.

```
#include <stdio.h>
main()
{
int x;
     while((x = getchar()) != EOF) {
          printf("%c",x);
     }
}
```

Notes:

- o Expect getchar to be a macro in most C libraries. It has no side effects.
- o Under CP/M, the BIOS will complain on a disk read error, which is the only possibility besides end of file. In this case, the system will kill the running program unless you insist upon continuing with the error.
- o A return of -1 under C/80 means that end of file was encountered. Other C systems will require that you check EOF with feof().
- o A normal use of getchar() may use the companion function ungetc(x,stdin). This function puts the input value x back onto the stream so that the next getchar() call receives x.
- o getchar() reads from stream stdin, which may in fact be a file due to re-direction. For console properties see CHmode().
- o Binary console mode (Cmode <> 1) turns off editing, ctrl-B processing, CR/LF translation and EOF return for ctrl-Z.

The getddir Function

Purpose:

Fills an array char *p[] with file names and sizes, using a wildcard mask to direct the file selection. Total memory use for 255 files is about 5k bytes. The programmer handles memory management.

Function Header:

```
int getddir(s,p,nmax)
char *s;                        Wildcards for file name.
char *p[];                      p[i] points to 17-byte area.
int nmax;                       Size of array p[].
```

Returns:

n	Number of files processed. n <= nmax.
p[i][0] - p[i][14]	Null-terminated filename in special format "A:FILENAME.EXT".
p[i][15]	Last extent, 0-255.
p[i][16]	Number of records in the last extent.

Example: Print a sorted directory with file sizes for any mixed combination of wildcards across any number of disk drives.

```
#define RECLEN 17
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *w; static int i,j,k,n; static unsigned total;
char s[129],*p[255],q[255*RECLEN];

  for(j=i=0;i<255;++i,j += RECLEN) p[i] = &q[j];
  total = n = 0; i = 1;
  if(argc==1) argv[++argc] = "";
  do {
     strcpy(s,argv[i]);
     if(index(s,'.')==0) strcat(s,"*.*");
     n += getddir(s,&p[n],255-n);
  }  while(++i<argc && n<255);
  ssort2(n,p,RECLEN);
  for(j=i=0;i<n;++i) {
       k = kilos(p[i]); total += k;
       printf("%-14s%3dk",p[i],k);
       if(++j==4) {putchar('\n'); j = 0;}
       else putchar(' ');
  }
  if(j) putchar('\n');
  printf("%d Files using %u kilobytes\n",n,total);
}
```

```
kilos(r) char *r;
{
  return (16*r[15]+(7+r[16])/8);
}
```

Notes:

o For CP/M 2.2xx only. Library hooks are documented in STDIO.H.

o To just print the file names, use puts(p[i]). The null terminator always appears before the extent and last record count.

o The file size is usually 128*recs+16384*extent. This will require long integer arithmetic in general. Ascii files mark the end with ctrl-Z (usually in the last record of the file).

o To get the file size in kilobytes, using only integer arithmetic, use 16*extent + (recs+7)/8.

o Extents of a file need not be consecutively located in the directory tracks. Further, two extents may be lumped into one entry. A full directory entry uses 16k bytes.

o The complete source for the function getddir follows. It is often the case that this program is engineered for a special application. Use the archives to get a working copy.

```
int getddir(s,p,nmax)
char *s;                      /* wildcards for file name */
char *p[];                    /* p[i] points to 17-byte area */
int nmax;                     /* size of array p[] */
{
static char *q;
static int i,n,drive;
char *findFIRST(),*findNEXT(),*decodF();

  if((q = findFIRST('?',s)) == (char *)-1) return 0;
  drive = (s[1]==':') ? ((s[0]|32)-'a') : (logged());
  n = 0;
  do {
       if(n >= nmax) break;
       q[0] = drive+1;
       strcpy(p[n],decodF(q));
       for(i=0;i<n;++i){
         if(strcmp(p[i],p[n])==0 && p[i][15]<q[12]) break;
       }
       p[i][15] = q[12];                /* extent */
       p[i][16] = (255 & q[15]);        /* records */
       if(i == n) ++n;
  } while((q = findNEXT()) != (char *)-1);
  return n;
}
```

Extended Directory Utility

Below is the source code for a sorted directory utility which is an extension of the example above. Source appears in the archives.

The features:

- o Displays sorted directory on any number of drives for any number of wildcard filespecs. Output to disk or printer via I/O re-direction.
- o Gives disk space report on command which includes group size and disk capacity.
- o Output can be in 1,2,3,4 columns.

Command strings can be as long as the CP/M command line. Here are some examples:

A>dd A: B: C:	Disk directories of drives A:,B:,C:
A>dd A:	4-column directory of drive A:
A>dd A: ;1	1-column directory of drive A:
A>dd A: ;2	1-column directory of drive A:
A>dd A: ;3	1-column directory of drive A:
A>dd A: ;	Directory of drive A: plus free space.
A>dd A:*.COM	Display all files ending with .COM
A>dd .com	Same as A:*.COM
A>dd A:f*.*	Display all files starting with f.
A>dd f	Same as A:f*.*
A>dd ?.	Display one-letter file names.
A>dd B:*.com ;	Display .COM files on B: plus free space.
A>dd B:*.C *.COM	All .C files and .COM files on B:.

A selected drive remains the default, as in the last example above. Free space reports are issued when encountered, therefore they will occur before file displays. A disk free space report causes a drive reset, in order to insure that disk changes are recognized.

```
#include <stdio.h>
kilos(r) char *r;
{
   return (16*r[15]+(7+r[16])/8);
}

struct dfree {
   unsigned avail;      /* free groups */
   unsigned total;      /* total groups */
   unsigned bpsec;      /* bytes per sector */
   unsigned spg;        /* sectors per group */
};
```

```
freespace(d)
int d;
{
char s[3];
unsigned kbytes; struct dfree df; unsigned getdfree();

    if(d==0) d = logged(); else --d;
    s[0] = d+'A'; s[1] = ':'; s[2] = 0;
    kbytes = getdfree(s,&df);
    printf("%c: %uk free in %d groups\n",
           s[0],kbytes,df.avail);
    printf("Disk capacity %dk using %d-byte groups\n",
           df.total*(df.bpsec*df.spg/1024),
           df.bpsec*df.spg);
}

#define RECLEN 17
main(argc,argv) int argc; char **argv;
{
static int i,j,k,n,w,drive; static unsigned total; int getddir();
char fcb[36],s[129],*p[255],q[255*RECLEN];

    for(j=i=0;i<255;++i,j += RECLEN) p[i] = &q[j];
    total = n = 0; i = 1; w = 4;
    if(argc==1) argv[++argc] = "";
    drive = logged();
    do {
        strcpy(s,argv[i]);
        if(index(s,'.')==0) strcat(s,"*.*");
        makeFCB(fcb,s);
        if(fcb[0]) drive = fcb[0]-1; else fcb[0] = drive+1;
        if(fcb[1]==';') {
            w = fcb[2]-'0'; if(1<=w && w<=4) continue;
            if(w == '?'-'0') { freespace(fcb[0]); continue;}
            w = 4;
        }
        if(fcb[1]==' ') fcb[1]='*';
        strcpy(s,decodF(fcb));
        n += getddir(s,&p[n],255-n);
    } while(++i<argc && n<255);
    ssort2(n,p,RECLEN);
    for(j=i=0;i<n;++i) {
        k = kilos(p[i]); total += k;
        printf("%-14s%3dk",p[i],k);
        if(++j >= w) {putchar('\n'); j = 0;}
        else putchar(' ');
    }
    if(j) putchar('\n');
    if(n) printf("%d Files using %u kilobytes\n",n,total);
}
```

The getdfree Function

==

Purpose:

Computes the number of free kilobytes on the disk. Return disk drive info. See below. Disk drive is entered as a string.

Function Header:

```
unsigned getdfree(drive,dfp)
char *drive;                          Drive, e.g., "A:"
struct dfree *dfp;                    See the example below.
```

Returns:

u Free disk space in kilobytes.

+ Additional information is returned in the 4-word structure passed as the second function argument. The array is organized as follows:

DESCRIPTION	ARRAY	STRUCTURE
Free group count	info[0]	dfp->avail
Full disk group count	info[1]	dfp->total
Bytes per logical sector	info[2] = 128	dfp->bpsec
Logical sectors per group	info[3] = 8 or 16	dfp->spg

Example: Print disk drive statistics for any drive.

```
struct dfree {
   unsigned avail;          /* free groups */
   unsigned total;          /* total groups */
   unsigned bpsec;          /* bytes per sector */
   unsigned spg;            /* sectors per group */
};
#include <stdio.h>
main()
{
unsigned kbytes;
char s[129];
struct dfree df;
unsigned getdfree();
   fputs("Enter disk drive letter: ",stderr);
   gets(s);
   kbytes = getdfree(s,&df);
   printf("%uk free on %c:\n",kbytes,s[0]);
   printf("%d free groups (group size = %d bytes)\n",
           df.avail, df.bpsec*df.spg);
   printf("Disk capacity %d groups\n",df.total);
}
```

getdfree

Notes:

- For CP/M 2.2xx only. Library hooks are documented in STDIO.H.

- The current version resets the disk system each time it is called. This feature logs in drive A: and also the drive to be accessed. All other disks are reset.

- The disk free space report has the additional feature of taking proper action for diskette changes. Reports the correct free space regardless of the number of disk swaps.

- Bad drive specs are mapped to drive A:. Some bad drive input can make its way to the program, in which case CP/M itself will abort.

- The complete source for the program getdfree follows. It is often the case that this program is altered and engineered for a particular application. See the archives to get a working copy. Below is for reference during brainstorming.

```
#define CURDISK         25          /* BDOS                 */
#define RESETDRIVES     13          /* BDOS                 */
#define LOGDISK         14          /* BDOS                 */
#define SELDSK          9           /* BIOS                 */
#define SECSIZ          128         /* Logical sector size  */

struct dfree {
        unsigned avail;         /* free groups       */
        unsigned total;         /* total groups      */
        unsigned bpsec;         /* bytes per sector  */
        unsigned spg;           /* sectors per group */
};

unsigned getdfree(drive,dfreep)
char *drive;
struct dfree *dfreep;
{
/* CP/M 2.x Disk Parameter Tables */

struct dpb {            /* Disk Parameter Block */
        unsigned spt;   /* Sectors/track */
        char bsh;       /* block shift (3 = 1k, 4 = 2k, 5 = 4k) */
        char blm;       /* a bit mask 'bsh' bits wide */
        char exm;       /* extent mask  */
        unsigned dsm;   /* maximum block number for drive */
        unsigned drm;   /* maximum directory entry index */
        char al[2];     /* Directory Allocation Bitmap */
        unsigned cks;   /* size of checksum vector */
        unsigned off;   /* Number of reserved tracks */
        } ;             /* pointer obtained from dphp->hdpbp */
struct dph {            /* Disk Parameter Header */
        char *xltp;     /* pointer to sector translation table */
        int dscr[3];    /* BDOS scratch */
        char *dirbp;    /* Pointer to directory buffer */
        struct dpb *hdpbp; /* Disk Parameter Block pointer */
        char *csvp;     /* Pointer to checksum vector */
        char *alvp;     /* Pointer to Allocation Bitmap */
        } ;             /* pointer returned by bios seldsk */

static
int maxblock,blk,freeblk,disk,cdisk,n;
static
char bitnumber[] = {
        128,64,32,16,8,4,2,1};
static
unsigned *u;
static
char *map;
static
struct dph *dphp;
static
struct dpb *dpbp;

    disk = ccBDOS(CURDISK);
    ccBDOS(RESETDRIVES);
    n = toupper(drive[0]) - 'A';
    cdisk = (0 <= n && n <= 15 && drive[1] == ':') ?
        n : disk;
    ccBDOS(cdisk,LOGDISK);
    dphp = ccBIOS(1,cdisk,SELDSK); /* pointer disk param header */
    dpbp = dphp->hdpbp;            /* pointer disk param block */
    maxblock = dpbp->dsm;          /* max block number, 0=first */
    map = dphp->alvp;              /* pointer to allocation bitmap */
    for(freeblk=blk=0;blk<=maxblock;++blk)
      if((map[blk/8] & bitnumber[blk%8])==0) ++freeblk;

    u = (unsigned *)dfreep;
    *u++ = /* dfreep->avail */ freeblk;
    *u++ = /* dfreep->total */ maxblock+1;
    *u++ = /* dfreep->bpsec */ SECSIZ;
    *u   = /* dfreep->spg   */ (SECSIZ << dpbp->bsh)/SECSIZ;

    ccBDOS(disk,LOGDISK);
    return ( (unsigned)((*u/(1024/SECSIZ))*freeblk) );
}
```

The getl Function

==

Purpose:

Reads a 32-bit long integer from the stream, in Intel Reverse Format.

Function Header:

```
long getl(fp)
FILE *fp;                Stream pointer, open stream
```

Returns:

-1	Error or end of file
n	32-bit word, otherwise

Example:

```
VOID getlongs(fp)
FILE *fp;
{
long x,getl();
int feof();
  while(1) {
     if((x = getl(fp)) == EOF) {
         if(feof(fp) == EOF) break;
     }
     printf("Long x = %ld\n",x);
  }
}
```

Notes:

- o To detect end of file, use feof() every time getl() returns a value of -1. Do not check end of file on every read - the overhead of feof() is too much.

- o Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. In addition, poorly constructed libraries may write getl() from getc(), which introduces the possibility of CR/LF translation.

- o Getl() calls the library primitive getcbinary() to avoid any character translation. Most Unix Ports to newer machines do not translate characters in files anyway. Code that uses getl() on an ASCII file should run without changes on other targets.

The getline Function

==

Purpose:

Reads a line of text from the console, or the re-direction file for stream stdin, up to and including the newline. The string is null-delimited.

Function Header:

```
int getline(s,n)
char *s;                      Base address of string buffer.
int n;                        Size of the buffer in bytes minus 1.
```

Returns:

EOF	If end of file was read before any other characters.
n	The number of characters read, which always includes a newline character for a successful read operation.

Example: Read a line of text from the console, print it.

```
#include <stdio.h>
main()
{
char buf[129];
int i;
     printf("Enter a line of text: ");
     i = getline(buf,128);
     if(i != EOF) {
       printf("Characters = %d\n",i);
       printf("Data = %s",buf);
     }
}
```

Notes:

- o Reading more than the allowed number of bytes without an intervening newline causes the excess characters to be lost.
- o Empty input returns 1, not 0, and buf[0] = '\n'.
- o At end of file, getline returns -1.

The getln Function

==

Purpose:

Reads a line of text from the console, or the re-direction file for stream stdin, up to the newline. The string is null-delimited. The newline is stripped from the buffer.

Function Header:

```
int getln(s,n)
char *s;                       Base address of string buffer.
int n;                         Size of the buffer in bytes minus 1.
```

Returns:

EOF	If end of file was read before any other characters.
n	The number of characters read, which always includes a newline character for a successful read operation.

Example: Read a line of text from the console, print it.

```
#include <stdio.h>
main()
{
char buf[129];
int i;
     fprintf(stderr,"Enter a line of text: ");
     i = getln(buf,128);
     if(i != EOF) {
       fprintf(stderr,"Characters = %d\n",i);
       fprintf(stderr,"Data = %s",buf);
     }
}
```

Example: Read lines of text from stream stdin, write to stdout.

```
#include <stdio.h>
main()
{
char buf[129];
     while(getln(buf,128) != EOF) puts(buf);
}
```

Notes:

- o Reading more than the allowed number of bytes without an intervening newline causes the excess characters to be lost.

- o Empty input returns 1, not 0, and buf[0] = '\0'. Return is the same as getline().

- o Overcomes the limitation of getline(), which is the extra newline.

- o Replaces the more dangerous and limited gets(), which can write over the data area due to long lines. But for up to 128 bytes of input, gets() and getln() are the same function.

- o This function is not in K&R nor is it in the Unix System III standard. It should be. Range and error checking are essential to the production of bullet-proof user interfaces.

- o Entry of ctrl-z returns 0. See K&R.

The getpass Function

Purpose:

Reads in a password without console echo.

Function Header:

```
char *getpass(q)        Returns a static storage location

char *q;                Prompt string to issue to console
                        before password is typed. Must be
                        null-terminated.
```

Returns:

Null-terminated password string pointer
to a static area of memory where the user
input resides. This area is 8 bytes long
plus one NULL. It is overwritten on each call.

Notes:

- o The standard ccBIOS(3) interface is used so that we read directly from CP/M. This prevents type-aheads.
- o The caller must map case and worry about things like delete and backspace.
- o The user's carriage return is echoed, and it happens automatically if the character limit is exceeded.

Example: Ask for user password "SPECIAL" and return 0 on success, -1 on failure.

```
VOID getmypassword()
{
#define MYPASSWORD "SPECIAL"
char *p,*getpass();
  p = getpass("Enter password: ");
  cvupper(p);
  if(strcmp(p,MYPASSWORD) == 0) return 0;
  return -1;
}
```

The getpid Function

Purpose:

Get the process ID of the current process.

Function Header:

```
int  getpid()
```

Returns:

Bogus value of 0, for Unix compatibility.

Notes:

o May not make a Unix program work, but it helps it to compile without errors.

The gets Function

===

Purpose:

Reads characters from open stream stdin and null-terminates the string. Breaks on newline or EOF. The newline is purged from the buffer.

Function Header:

```
char *gets(s)
char *s;                  Base address of the storage area.
```

Returns:

```
(char *)0                 End of file and string empty.
(char *)0                 Stream error and partially filled string.
(char *)s                 Success, base of storage area.
```

Example: Read lines from stdin and print until end of file. Note the stripped linefeed and the extra newline on output.

```
#include <stdio.h>
main()
{
char s[129];
     while(TRUE) {
          fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
          if(gets(s) == (char *)0) break;
          fprintf(stderr,"%s\n",s);
     }
}
```

Notes:

- o It is impossible to tell whether a null pointer return stands for an error or end of file.
- o The array s[] may be filled with invalid data in case of an error.
- o Newlines read from the stream are NOT appended to the array s[]. This is different from fgets(), which appends the newline on input. In practise, it means puts() will output exactly the input received by gets().
- o There is no way to protect the storage area s[] from long lines when using gets(). It is best to use fgets() in the cases where over-run is possible.
- o Under C/80 and this library, gets() is a call to fgets() with buffer size 128. The newline is stripped off upon return from fgets(). This affords users some protection against crashes at the expense of a limited buffer size.

The getw Function

===

Purpose:

Reads a 16-bit word from the stream, in Intel Reverse Format.

Function Header:

```
unsigned getw(fp)          Unsigned integer, 16 bits
FILE *fp;                  Stream pointer, open stream
```

Returns:

-1	Error or end of file
n	16-bit word, otherwise

Example: Read and print integers taken from an open binary file.

```
VOID getwords(fp)
FILE *fp;
{
int x,getl();
int feof();
  while(1) {
    if((x = getl(fp)) == EOF) {
        if(feof(fp) == EOF) break;
    }
    printf("Word x = %u\n",x);
}
```

Notes:

o To detect end of file, use feof() every time getw() returns a value of -1. Do not check end of file on every read - the overhead of feof() is too much.

o Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. In addition, poorly constructed libraries may write getw() from getc(), which introduces the possibility of CR/LF translation.

o Getw() calls the library primitive getcbinary() to avoid any character translation. Most Unix Ports to newer machines do not translate characters in files anyway. Code that uses getw() on an ASCII file should run without changes on other targets. Beware of size restraints: getw() is 16 bits, but on 36-bit machines it may actually be 18 bits (9 bits per byte instead of 8).

The heapsort Function

Purpose:

Heapsort for an array of character strings with easy-to-change comparison method. Changes pointers but not actual data in the string.

Function Header:

```
VOID heaps(base,n,cmp)
char *base[];           Base address of pointer array
int  n;                 Number of array elements to sort
int  (*cmp)();          Compare routine for two strings p,q
                        that returns an integer with the
                        same rules used by Unix strcmp(p,q):

                           = 0    strings p,q are equal
                           < 0    p < q
                           > 0    p > q
```

Returns:

Nothing of value

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
        heaps(p,10,strcmp);
        for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a              Numerals are less than letters
1. 1, C              in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3              UPPERCASE is less than lowercase
8. a, 8              in the ASCII standard
9. d, 5
```

Notes:

- o You can most often use strcmp() as the argument for cmp().

- o It is necessary to declare the function cmp() before the call to heaps(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.

- o To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example below.

- o This source file is very portable. Use it on any system with minimal C compiler.

Reference:

A C Reference Manual, pp 62-63,
by Harbison & Steele
Prentice-Hall, Englewood Cliffs (1984)

The highmem Function

Purpose:

Computes the highest memory address that is assignable by sbrk().

Function Header:

```
char *highmem()
```

Returns:

The address below the stack which is the last possible address assignable by sbrk(). This implementation assumes 600 bytes of stack space, which is used in the computation in order to give a conservative estimate.

Example: Find out how many bytes of free contiguous memory are available. Do it three ways.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char *highmem(),*lowmem();
unsigned amt,memsize();
     amt = highmem() - lowmem();
     printf("%u bytes available for use by sbrk()\n",amt);
     amt = ((char *)(&argc) - (char *)sbrk(0));
     printf("%u bytes available for use by sbrk()\n",amt);
     amt = memsize();
     printf("%u bytes available for use by sbrk()\n",amt);
}
```

Notes:

o The 600 bytes of slop is a guess. The right amount depends on your program.

o The value returned from highmem() depends on its location in the calling program. If called from main(), then the answer is accurate. If buried deeply in subroutine calls, the answer could be very optimistic.

o The method used for finding out about heap space **works on 8080 machines** with CP/M 2.2. It **will not work with** CP/M 3.0. It **will not generalize to mainframes or 8086-type CPU machines. It does work on a Motorola 68000 with CP/M-68K.**

The Horner Function

==

Purpose:

Compute a polynomial value by Horner's method..

Function Header:

```
float Horner(x, p, n)
float x;                    Polynomial variable x value.
float p[];                  Coefficients of polynomial.
int n;                      Degree of the polynomial.
```

Returns:

The computed float value obtained by putting x into the polynomial equation.

Example: Given p[2] = 3, p[1] = 4, p[0] = 6, then 3*(x**2) + 4*x + 6 = x(x(3) + 4) + 6 = x(x(p[2]) + p[1]) + p[0]. The degree of the polynomial is 2 and the factorization scheme is known as Horner's method.

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
static float p[] = {
     6,4,3
};
float atof();
float Horner();
     printf("Enter a float: "); gets(s); f = atof(s);
     printf("For x = %f, 3*(x**2) + 4*x + 6 = %f\n",
             x,Horner(x,p,2));
}
```

Notes:

o No error checking. This function uses the standard factorization method taught in college algebra.

o To understand the method, study the above example carefully. Note in particular the dimension of the array p[] versus the degree of the polynomial generated from the coefficient array.

The index Function

Purpose:

Finds the character position of a byte inside a string.

Function Header:

```
char *index(str,c)
char *str;                      Source string.
char c;                         Character to locate.
```

Returns:

&str[i]	Where c == str[i], on success.
(char *)0	Failure.

Example: Find the colon in a file spec, strip off the file name, print.

```
#include <stdio.h>
main()
{
char s[129],t[129],*p;
     printf("Enter a file name: ");
     gets(s);
     cvupper(s);
     if(p = index(s,':')) strcpy(t,p+1);
     else strcpy(t,s);
     printf("Stripped file name: %s\n",t);
}
```

Notes:

o Usage of this function differs among libraries. The Unix V7 standard, according to Bourne, is the above.

o Index() should be capable of locating the null terminator in a string. For portability, we recommend using p + strlen(p) as a pointer to the end of a string.

o Index() is the same function as **strchr()**. See Harbison & Steele.

The inport Function

Purpose:

Reads a value from a given CPU port.

Function Header:

```
int inport(port)
int port;                     Port number.
```

Returns:

The value read from the port. Assumes data was available. See system notes below and the example.

Example: Read characters from the modem port.

```
#define MODEM      0330   /* Z90A computer, Z80 CPU, 255 ports */
#define OFFSET     0005   /* 8250 UART, line status register */
#define READY      0001   /* test data ready at port         */
#include <stdio.h>
main()
{
     while(TRUE) {
          while((inport(MODEM+OFFSET)&READY)==0 &&
                 bdos(11,0)==0) ; /* wait for keyboard or modem */
          outport(MODEM,x);                  /* ship the byte    */
     }
}
```

Notes:

- On most machines the status of the port must be interrogated in order to read from the data port. Buffered ports may require a different method, namely checking the character count in the buffer.

- Disk I/O ports are often interrupt driven. Reading from the port will require set-up of the interrupt vector for return.

- Clock and timer chips like the 8253 Intel make for good examples of port usage in C programs. See also Analog-to-Digital boards and parallel port interfaces. These appear on music boards and game interfaces. The signals are joysticks, mice, light pens, plotters, graphics tablets and the like.

- Port operations are inherently machine-dependent. Don't expect to write portable code. But do expect READABLE code. Watch out for 8-bit and 16-bit ports on other machines. Not all ports are 8-bits wide.

The insert Function

==

Purpose:

Inserts a CP/M command tail into the two default file control blocks and then copies the command tail to the CP/M default buffer.

Function Header:

```
     VOID insert(tail)
     char *tail;                 CP/M command tail to be inserted.
```

Returns:

Nothing. Similar to the insert command in the DDT debugger.

Example: Swap in a new command tail into the default buffers prior to chaining to a new overlay. The link program LEXI.COM takes a command line with three file name arguments.

```
     #include <stdio.h>
     main()
     {
     extern char *Cbuf;          /* C console buffer is 136 bytes */

          insert(" TEST1.TXT TEST2.TXT OUTPUT.TXT");
          makeFCB(Cbuf,"LEXI");
          chain(0x100,Cbuf);
          puts("LEXI not found");
     }
```

Notes:

- o Multiple arguments may be used. The command line limit of 128 characters must be obeyed, however.

- o All characters are acceptable, however uppercase translation is performed to simulate CP/M handling of command lines.

- o The string may contain whatever you wish.

- o CP/M 2.2 might demand in certain applications that the information found in the default buffers also be present in the CCP buffers. This is highly unusual, but may be a problem.

- o The file control blocks are built at BOOT+0x5C and BOOT+0x6C. Tokens for these file control blocks are the first two found in the command tail.

The instr Function

==

Purpose:

Computes the offset position for a substring match.

Function Header:

```
int instr(n,strl,str2)
int n;                          Character position to start.
char *strl,*str2;               Source strings, null-terminated.
```

Returns:

0	No substring match
k	Substring of strl matched str2 at position k.

Example: Find the word SAILOR inside a sentence typed by the user.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int k;
     if(argc <= 1) exit();
     if(k = instr(1,argv[1],"SAILOR"))
          printf("Word SAILOR found at position %d\n",k);
     else
          printf("Word SAILOR not found in string\n");
}
```

Notes:

- o Instr() is similar to the BASIC function of the same name. It takes three arguments. You may not drop arguments (as is done in BASIC).

- o The value of instr(1,"abcd","abc") is 1, not zero.

- o The value of instr(1,"xabcdabcd","abc") is 2 whereas the value of instr(3,"xabcdabcd","abc") is 6. Counting of the offset index starts at 1. The maximum offset is the string length of strl.

- o This function also works with unsigned integers.

The IOTABLE module

Purpose:

Storage area for I/O information, channel data. Used by file routines such as read() and write(), seek(), getc(), putc(), fopen(), fclose(), fflush(), ffb(), fseek(), ftell().

VARIABLE LIST

MAXCHN — Equals 1+n, where n is the maximum number of disk files. Settable by re-assembly to any number from 1 to 251. Currently, n = 6 and MAXCHN = 7.

DEVICES — Numbers 252,253,254,255 are reserved for CON:, RDR:, PUN:, LST: respectively. Number 0 is translated by the software into the console number 252.

char *IObuf[MAXCHN];	Addresses of the file buffers
int IOsect[MAXCHN];	Sector count in 128-byte hunks
char IOrw[MAXCHN];	0 or 1 for last operation, read or write
int IOtmp;	Storage for buffer address, work variable
char IOch[MAXCHN];	Channel numbers (-1 means empty).
int IOind[MAXCHN];	Offset into file buffer for next char
char IOmode[MAXCHN];	Mode byte 'r', 'w', 'u'
char IObin[MAXCHN];	Binary flag '\0' or 'b'
char *IOfcb[MAXCHN];	File control block addresses
char IOpchan[4];	Physical device numbers 252,253,254,255
char IOpread[4];	Physical read CP/M function numbers 1,3,0,0
char IOpwrit[4];	Physical write CP/M function numbers 2,0,4,5
char IOpeof[4];	Physical device EOF characters 26,26,26,12
char *IOdev;	Physical device string 'con:rdr:pun:lst:' with null terminator, all lowercase.
int IOnchx[MAXCHN];	Amount in bytes read by ffb() into buffer
int IOend[MAXCHN];	Saves EOF record number for seek
int IOsize[MAXCHN];	IO buffer sizes (0 for chan 0, 256 default)

The IOTABLE module

==

The first **WARNING** to be issued is that offset 0 is not used in any of the arrays. All descriptors start at offset 1 or higher but never exceed MAXCHN-1.

The functions of the I/O library access the IOTABLE, and use the module END.CC, which contains the end-of-program label. This module is pure data (no code segment). It holds the secret to understanding the method by which dynamic buffers are handled in the C library.

Pre-defined file buffers appear in memory as though they were assigned by sbrk(), which means they are located immediately after the physical end of the program. Walt Bilofsky's library puts them into high memory, which is unsuitable for buffer re-allocation and recovery of buffer space via sfree(). The placement of the file buffers in this library allows for programmable buffer sizes and buffer locations.

File buffer size defaults to the size pre-set in STDIO.H. We recommend a buffer size of 256 for C/80 library compatibility. To obtain Unix compatibility, use default buffer size 512.

The size of a file buffer is set by the symbols $SIZ1, ... , $SIZ6. These are assembly language equates that are recognized by IOTABLE. The number of symbols equals the number of possible open files, MAXCHN-1. In order to change the number of files beyond 6, you must edit IOTABLE.CC, re-assemble and invoke LIB.COM to insert the new module into the library.

The macro variable nfiles defined in STDIO.H allows you to select the number of files from 1 to MAXCHN-1. This setting has an effect on the amount of heap space.

The macro variables bufsz1, bufsz2, bufsz3 defined in STDIO.H are used for the purpose of selecting the buffer sizes. The interface allows file buffer size definition directly from language C, in a convenient fashion that is self-documenting.

File control blocks are assigned as needed as though sbrk() were called. Each file control block consumes 36 bytes. File control blocks with IOfcb[fd]=0 will force a call to sbrk() to get space. The construction of file control blocks is handled through routines in XFCB.C, which supports wildcard expansion and uppercase translation, plus white space skipping.

The IOTABLE module

===

The essential arrays in IOTABLE.CC are IOfcb[], IObuf[], IOsize[]. The address of the file control block is saved in IOfcb[fd]. If the contents of IOfcb[fd] is zero, then fopen() will call sbrk() for 36 bytes of space and update IOfcb[fd] accordingly. Similarly, fopen() checks IObuf[] for a nonzero value, and calls sbrk() for IOsize[fd] bytes as needed, with IObuf[fd] updated to reflect the new buffer address.

To fool the system, you must change IOsize[fd] at a time when the file buffer is empty. The safest method is to rewind the file or change when the file is open and not yet accessed. You can always fool with the buffers and sizes when the channel is unused, but there is no guarantee that an fopen() call will return that channel.

Changes to IOsize[] must be matched with a change of IObuf[]. If the stream is unused, then it is safe to set IObuf[fd]=IOfcb[fd]=0 and call sfree() to free up the buffer space for re-use. A later call to fopen() will automatically get the desired space from sbrk().

As you may have guessed, all the file buffer and file control block space can be recovered by closing all the files and setting IOfcb[] = IObuf[] = 0 for all MAXCHN-1 channels. Use sfree(-1) to get all of memory back. Beware: this invalidates all previous core() and sbrk() function calls!.

The isatty Function

Purpose:

Tests a file descriptor fd to see if it is attached to the console device.

Function Header:

```
int  isatty(fd)
int  fd;              File descriptor fd (as distinguished from a
                      stream pointer fp). Use fileno() to obtain fd
                      from a stream pointer:

                        fd = fileno(fp);
```

Returns:

0	fd not attached to console
1	fd is mapped to the console device

Example: Test re-direction channel stdin to see if it is set to the console.

```
#include <unix.h>
#include <stdio.h>
VOID whichone()
{
int fd;

    fd = fileno(stdin);
    if(isatty(fd)) puts("Input is from the console");
}
```

Notes:

o The library uses fd = 252 for the console.

o However, 0 is mapped to the console too. For compatibility with defaults, the special value fd = 0 returns isatty = 1.

The isgraph Function

Purpose:

Test for a printable character in the range 041 to 0176 (octal).

Function Header:

```
int  isgraph(c)
char c;                  Character to test
```

Returns:

```
    0       Not a graphic character
 NONZERO    Is a graphics character
```

Source code: This function is so often ill-defined and wrongly used that we include the source code to communicate the precise difference between this function and its popular aliases.

```
int isgraph(c)
char c;
{
     if(c == ' ') return 0;
     return isprint(c);
}
```

Notes:

o This range includes all characters from isprint() except SPACE itself.

The istype Functions

Purpose:

The purpose of this group of tools is to classify and test individual characters for inclusion in a particular class.

Function Header:

```
int isascii(c) - Tests c for the range 0 to 127 decimal.
int isalnum(c) - Tests c for alphabetic or numeric.
int isalpha(c) - Tests c for alphabetic.
int isupper(c) - Returns -1 for uppercase c, else 0.
int islower(c) - Returns -1 for 'a' <= c <= 'z', else 0.
int isxdigit(c) - Tests for a valid hexadecimal digit, 0-9,A-F.
int isdigit(c) - Tests for a valid digit, 0-9.
int isspace(c) - Tests for blank, tab, carriage return, linefeed.
int iscntrl(c) - Tests for a control character 0-31 decimal.
int isprint(c) - Tests for a printable character.
int ispunct(c) - Tests for punctuation (not ctrl or alphanumeric).
```

Throughout, c is assumed to be a character, however it may be an integer variable instead. These routines ignore sign extension.

Returns:

```
-1  -  TRUE            The K&R standard is NONZERO return.
 0  -  FALSE
```

Example: Access istype functions by the array method.

```
#include <unix.h>
#include <ctype.h>
#include <stdio.h>
main()
{
/* same example as below, but uses macros and the array method */
}
```

Example: Access standard library functions.

```
#include <stdio.h>
main()
{
int c;
char buf[129];
   while(TRUE) {
     printf("Enter a character: ");
     gets(buf);
     c = buf[0];
     printf( isascii(c) ?
             "isascii(c) = TRUE" : "isascii(c) = FALSE");
     printf( isalnum(c) ?
             "isalnum(c) = TRUE" : "isalnum(c) = FALSE");
     printf( isalpha(c) ?
             "isalpha(c) = TRUE" : "isalpha(c) = FALSE");
     printf( isupper(c) ?
             "isupper(c) = TRUE" : "isupper(c) = FALSE");
     printf( islower(c) ?
             "islower(c) = TRUE" : "islower(c) = FALSE");
     printf( isxdigit(c) ?
             "isxdigit(c) = TRUE" : "isxdigit(c) = FALSE");
     printf( isdigit(c) ?
             "isdigit(c) = TRUE" : "isdigit(c) = FALSE");
     printf( isspace(c) ?
             "isspace(c) = TRUE" : "isspace(c) = FALSE");
     printf( iscntrl(c) ?
             "iscntrl(c) = TRUE" : "iscntrl(c) = FALSE");
     printf( isprint(c) ?
             "isprint(c) = TRUE" : "isprint(c) = FALSE");
     printf( ispunct(c) ?
             "ispunct(c) = TRUE" : "ispunct(c) = FALSE");

   }
}
```

Notes:

- o Most libraries implement these functions as macros which access an array of character classes. See CTYPE.H and related documentation. The array method has a fixed overhead and portability going for it at the expense of code size (in some cases).

- o All functions above are honest functions and not macros. There are no side effects, even for the ctype functions.

- o If you use only isspace(), then including CTYPE.H will cost you dearly in code size. For a few istype functions it is more economical to use the standard functions in CLIB.REL.

The itoa Function

==

Purpose:

Converts an integer into a null-terminated string of decimal digits, with leading minus sign as appropriate.

Function Header:

```
     char *itoa(x,s)
     int x;                     Integer to be converted.
     char *s;                   Buffer for string of decimal digits.
```

Returns:

```
     s                    Base address of the conversion string.
```

Example: Convert a data item to a string of digits and print.

```
     #include <stdio.h>
     main()
     {
     char s[20];
     #define NUMBER 10133
          itoa(NUMBER,s);
          puts(s);
     }
```

Notes:

- o Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is imposed on the number.

- o The internal storage of an integer is two bytes (16 bits), stored in the usual 8080 reverse format. Integers typed at the terminal are in ascii decimal format, using the character set '0',...,'9'.

- o No check is made to see if the storage area is large enough to accept the string . This is left to the programmer.

- o Do not confuse this function with ltoa(), which converts a long signed integer to ASCII. The latter uses a 32-bit first argument.

- o Numbers larger than 32767 are assumed to be long integers by the compiler. If you want to convert an unsigned integer to ASCII, then use the function sprintf() instead.

The keystat Function

==

Purpose:

Test the console buffer and the keyboard for a character available.

Function Header:

```
int keystat()
```

Returns:

0	No character is available.
nonzero	Character is available.

Example: Expunge the console buffer.

```
main(argc,argv) int argc; char **argv;
{
char s[129];
  puts("Press any key to start");
  getbyte();
  conflush();
  puts("\nEnter a line of text");
  gets(s);
  puts(s);
}

conflush()
{
  while(keystat()) getbyte();
}
```

Notes:

o Function keys can create havoc with character mode input. Use keystat() to tell whether or not a character is available.

o This keystat() is not the same as bdos(11,0). In addition, it checks the console type-ahead buffer at Cbuf.

o Scanf() can cause problems too. We suggest that you flush the console input buffer as in the example each time that scanf() fails to return the required number of arguments.

The labs Function

==

Purpose:

Compute the absolute value of a signed long integer quantity.

Function Header:

```
long labs(value)                    Long Integer Absolute value
long value;                         Long argument, 32 bits
```

Returns:

The absolute long integer value of the argument.

Example: Print the long integer absolute value of a number entered at the console.

```
#define pfLONG 1
#include <stdio.h>
main()
{
char s[129];
long a,labs();

     printf("Enter long integer a: ");
     gets(s); a = atol(s);
     printf("labs(%ld) = %ld\n",a,labs(a));
}
```

Notes:

- o Not a macro. It often is a macro in other libraries.

- o There are no side effects. Labs() is an honest function.

- o The cast must appear explicitly in your program. For example, if you use labs(), then include a declaration of the form

  long labs();

- o A common error is to write ln = abs(n) where ln,n are signed long integers. This has undefined results, but there is no compile-time error reporting.

The ldexp Function

==

Purpose:

Computes x*(radix**n) where n is an integer and the value of x is typically 0.5 <= x < 1.0.

Function Header:

```
float ldexp(x,n)
float x;               Float value in range 0.5 to 1.0
int n;                 Integer exponent. Range -38 to +38 expected.
```

Returns:

```
x*(2**n)       Mantissa times power-two exponent.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
int n;
float x;
extern int errno;
float atof();
float ldexp();
  errno = 0;
  printf("Enter a float x: "); gets(s); x = atof(s);
  printf("Enter an integer n: "); gets(s); n = atoi(s);
  printf("ldexp(%f,%d) = %f, errno = %d\n",x,n,ldexp(x,n),errno);
}
```

Notes:

- o The float exponent in this library is r = ((int)255 & (int)x.c[3]) - (int)128.

- o A portable version of ldexp() that can be used to check an implementation appears below. The version in this library plugs the exponent in the float with a new value.

```
float ldexp(x,n) float x; int n;
{
int i; float f;
  i = n; f = x;
  if(i < 0) while(i++ < 0) f = f/2.0;
  else      while(i-- > 0) f += f;
  return f;
}
```

log and ln

The natural log Function
==

Purpose:

Compute the logarithm base e of real float value x.

Function Header:

```
#include <math.h>
float ln(x)          Log base e. Defined in MATH.H as log().
float log(x)         Log base e, or Naperian Logarithm.
float x;             Positive float argument.
```

Returns:

number	x in range
INF	x too large positive
-INF	x = 0 or x < 0
	errno = EDOM returned to flag error

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float log();
     errno = 0;
     printf("Enter positive float x: "); gets(s); x = atof(s);
     printf("log(%f) = %f, errno = %d\n",x,log(x),errno);
}
```

Notes:

o Method used is ln(x) = log10(x)/log10(e) = log10(x)*ln(10).

o Not very well done. Errors must filter back through the log10() function.

The log10 Function

Purpose:

Compute the base 10 logarithm of real float value x.

Function Header:

float log10(x)	Log base 10, same as log(x) on most calculators.
float x;	Positive float argument.

Returns:

number	x in range
INF	x too large positive
-INF	x = 0 or x < 0 errno = EDOM returned to flag error

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float log10();
     errno = 0;
     printf("Enter positive float x: "); gets(s); x = atof(s);
     printf("log10(%f) = %f, errno = %d\n",x,log10(x),errno);
}
```

Notes:

- o Method used is Rational approximation. See C&H 6.3.28. Constants from table 2325.

The logged Function

Purpose:

Reports the currently logged disk.

Function Header:

```
int logged();
```

Returns:

```
n               n = 0 to 15 for drives A: to P:
```

Example: Find out which disk drive is the default disk.

```
#include <stdio.h>
main()
{
int c;
  c = 'A'+logged();
  printf("Default drive is %c:\n",c);
}
```

Notes:

- Logged() is not a standard function. It is useful for writing interactive CP/M software.

- To simulate logged() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>
logged()
{
   return bdos(25,0);
}

#include <stdio.h>
logged()
{
   return ccBDOS(25);
}
```

The lsearch Function

==

Purpose:

Linear search and update of an arbitrary table of info. Requires a special compare function to match the internal table structure.

Function Header:

```
     char *lsearch(key,base,n,w,cmp)
     char *key;                    Key for table search.
     char *base;                   Base address of sorted table.
     int *n;                       Pointer to table size.
     int w;                        Width of each table element.
     int (*cmp)();                 Compare function cmp(key,tbl)

     int cmp(key,tbl)·             User-supplied compare routine.
     char *key;                    key = address of the key
     char *tbl;                    tbl = address of table entry;
```

Returns:

```
     (char *)       Location of table match
     (char *)0      Not found
```

Example: Search and update a list of strings.

```
#include <stdio.h>
     static char *tbl[] = {
       "ABC","AB","EFG","HJK","M","abc","abd","bcf",
       "","","","","","","","","","","","","","","",""
     };

     cmp(p,q) char *p,*q;
     {
       return strcmp(p,peekw(q));
     }
     main()
     {
     char key[129],*p,*q;
     int n,m,r; char *lsearch();
       m = n = 8;
       while(n < 24) {
         printf("Enter a string: "); gets(key);
         p = lsearch(key,tbl,&n,2,cmp);
         if(n>m) {
                pokew(p,q=core(strlen(key)+1));
                strcpy(q,key); m = n;
         }
         r = ((unsigned)p - (unsigned)tbl)/sizeof(char *);
         printf("%d items, tbl[%d] = %s\n",n,r,tbl[r]);
       }
     }
```

Notes:

- o The table width is 2 = sizeof(char *). The strings are not all of the same width. The table tbl[] is a table of pointers, which causes the unusual addressing modes seen in the example. Extra null pointers were used for table expansion. The code must test availability of expansion space.

- o This is a UNIX function. See also bsearch(), binary(), sbinary(), ssort2(), ssort3(), numsort(), dsort(), dsort16(), shells(), qsort(), quick(), heap().

- o The example shows how to make permanent entries into an updated table. Sometimes this is not necessary, especially for numerical tables.

- o The addressing modes required for string handling are unnatural. See the example above to get it right.

- o The complete source follows. The most common application error involves the integer pointer n.

```
char *lsearch(key,base,n,w,cmp)
char *key;
char *base;
int  *n;
int  w;
int  (*cmp)();
{
static
int m,j;
#define TARGET base
#define SOURCE &key
#define LENGTH W
     m = *n;
     while(m-- > 0) {
       if((j = (*cmp)(key,base)) == 0)
         return base;
       base += w;
     }
     *n = (*n)+1;
     moveMEM(TARGET,SOURCE,LENGTH);
     return TARGET;
}
```

The loginv Function

==

Purpose:

Fetches the logged-disk vector, a 16-bit flag vector.

Function Header:

```
unsigned loginv()
```

Returns:

n	A 16-bit word, where bit i is set when drive i is logged. Drives 0 to 15 correspond to A: to P:.

Example: Find out which disk drives are currently logged in.

```
#include <stdio.h>
main()
{
int i;
unsigned n;
  n = loginv();
  for(i=0;i<16;++i) {
    if(n&1) printf("Drive %c: is logged in\n",i+'A');
    n = (n >> 1);
  }
}
```

Notes:

- o Loginv() is not a standard function. It is important for writing interactive CP/M software.
- o To simulate loginv() on other systems, use the ccBDOS() function:

```
#include <stdio.h>
unsigned loginv()
{
unsigned ccBDOS();
   return ccBDOS(24);
}
```

The lowmem Function

==

Purpose:

Computes the lowest memory address that is assignable by sbrk(). The answer returned is sbrk(0).

Function Header:

```
char *lowmem()
```

Returns:

The break address for the program, last assigned by sbrk(), or fixed at compile time by the program text and data, plus the run-time file buffers.

Example: Check a file open to see if it called sbrk().

```
#define nfiles 0
#include <stdio.h>
main()
{
char *lowmem();
char *p;
FILE *fp,*fopen();
     p = lowmem();
     if(fp = fopen("TMP","w")) {
       if(p == lowmem()) printf("No change in sbrk()\n");
       else printf("sbrk() was called during fopen()\n");
       fclose(fp);
       unlink("TMP");
     }
}
```

Notes:

o This function is left around as an artifact. To produce portable code, use sbrk(0) or core(0) instead.

The ltoa Function

===

Purpose:

Converts a 32-bit signed long integer into a null-terminated string of decimal digits, with leading minus sign as appropriate.

Function Header:

```
char *ltoa(x,s)
long x;                         Long integer to be converted.
char *s;                        Buffer for string of decimal digits.
```

Returns:

```
s                   Base address of the conversion string.
```

Example: Convert a data item to a string of digits and print.

```
#include <stdio.h>
main()
{
char *ltoa();
char s[20];
#define NUMBER (long)1010133
#define UNUMB  (long)100
     puts(ltoa(NUMBER,s));
     puts(ltoa(UNUMB,s));
}
```

Notes:

- o Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is imposed on the number.
- o Signed integers from -32767 to 32767 will require a cast of (long) in order to be used as an argument to ltoa(). See the example.
- o Long integers are 32 bits (4 bytes). Storage of a long integer is such that you may address its lower order 16 bits as an integer, without error. However, this is highly non-portable (68000 CPU, for example).
- o The cast of ltoa() is (char *), which must be declared prior to usage as in the example above.

The makeFCB Function

==

Purpose:

Makes a CP/M file control block of 36 bytes as per interface standards. Expands * and ? wildcards.

Function Header:

```
VOID makeFCB(fcb,filename)
char fcb[36];                        File control block buffer.
char *filename;                      Name of file, null-terminated.
```

Returns:

Nothing useful. Fills the file control block with nulls, expands any wildcards * or ? in the file name to all question marks.

Example: Make a file control block from a wildcard file name.

```
#include <stdio.h>
main()
{
char fcb[36];
char *decodF();
  makeFCB(fcb,"A:*.TXT");
  printf("string = %s\n","A:*.TXT");
  printf("fcb = %s\n",decodF(fcb));
}
```

Notes:

- o Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.
- o Unspecified areas in the filename and extension are blank-filled. Areas beyond the file name are filled with nulls as per CP/M interface standards. A 36-byte file control block is assumed.
- o Classic mistakes in using makeFCB() include reversal of the arguments and the disaster of using a file control block shorter than the required 36 bytes.

The malloc Function

==

Purpose:

Malloc() allocates contiguous memory located between the end of the program and the stack. Free() releases memory assigned by malloc().

Function Header:

```
     char *malloc(m)
     unsigned m;                 Unsigned integer, amount of memory
                                 that is requested, in bytes.
```

Returns:

The base address of the area, on success, 0 if the request fails.

Example: Get 2048 bytes from malloc(), then free it.

```
     #include <unix.h>
     #include <stdio.h>
     getmem()
     {
     char *p;

          p = malloc(2048);
          if(p == (char *)0) puts("Request failed");
          else
          free(p);
     }
```

Structure used by malloc(): The following 4-byte header appears just before the base address returned by malloc(). It is used by both malloc() and free(), so be careful not to corrupt it.

```
       struct block {
               struct block
                   *nxtblk;
               unsigned
                      siz;
       };
```

Notes:

o Uses sbrk() to get raw system memory.

o You can't ask for more than 65531 bytes.

o Use free() to release a block.

o Code below derived from K&R(1978), pp 174-177. A few mods were made to compact the code under C/80.

o Two successive calls to malloc() will not in general result in a single block of continuous memory.

o Calls to malloc() cause a header to be written at the beginning of the block. Malloc() uses the header in an essential way - don't write over it!

o Sfree() doesn't know about malloc(). Look out!

o Brk() doesn't know about malloc(). Look out!

o Memory obtained from sbrk() comes in hunks of size HEAPSIZE:

```
#define SIZBLOCK        sizeof(struct block)
#define HEAPSIZE        256*SIZBLOCK
```

This implies that malloc() can fail even though sbrk() can assign more memory.

The max Function

==

Purpose:

Computes the maximum value of two integer arguments.

Function Header:

```
int max(x,y)
int x,y;                  Integers to compare.
```

Returns:

The larger of x and y as signed integers, i.e., max(-1,-2) = -1 and max(1000,2000) = 2000.

Example: Print the maximum value of two numbers a,b entered at the console.

```
#include <stdio.h>
main()
{
char s[129];
int a,b;

    printf("Enter number a: ");
    gets(s); a = atoi(s);
    printf("Enter number b: ");
    gets(s); b = atoi(s);
    printf("max(%d,%d) = %d\n",a,b,max(a,b));
}
```

Notes:

o The max() function used here is an honest function that works only on integers. It fails on long integers and floats.

o For long integers, use max = (a > b ? a : b).

o For floats, use max = (a > b ? a : b).

o Most libraries assume the max() function is a macro defined in the STDIO.H header file or in MATH.H. Such functions definitely have side effects. Beware when you port the code. Engineer against side effects when you write it for the first time.

The memsize Function

==

Purpose:

Computes the amount of free memory in the heap that is assignable by the next call to sbrk().

Function Header:

```
unsigned memsize()
```

Returns:

Unsigned integer in the range 0 to 65635, which represents the number of bytes free in the heap between the program end (as set by brk() or sbrk()) and the top of the user stack: memsize() = highmem()-lowmem().

Example: Find the size of the biggest possible text buffer and use it to load a file into memory with raw read.

```
#define nfiles 1
#define fd     fp            /* fd = fileno(fp), see fileno() */
#include <stdio.h>
main()
{
unsigned amt;
char *core();
FILE *fp,*fopen();
     amt = memsize();
     p = core(amt);
     printf("Internal buffer size %u\n",amt);
     if(fp = fopen("TMP","w")) {
       amt = read(fd,p,amt);
       printf("Read %u bytes into the internal buffer\n",amt);
     }
}
```

Notes:

- o This function is left around as an artifact. It helps to build a portable interface that allows code to be transported easily. Note that memsize() can be written for another system to return a fixed amount like 110000.

The midstr Function

==

Purpose:

Finds the position (1...32767) at which str2 matches str1.

Function Header:

```
int midstr(str1,str2)
char *str1;                  Source string.
char *str2;                  String to match.
```

Returns:

```
k>0       If str2 matches str1 starting at offset k-1.
0         No match.
```

Example: Test program for properties of midstr(). Reports timing differences between midstr() and instr().

```
#include <stdio.h>
main()
{
int i,j;
char str1[100],str2[100];
 for(;;) {
 fputs("Enter str1: ",stderr);
 gets(str1);
 fputs("Enter str2: ",stderr);
 gets(str2);
 timer(0);
 for(j=0;j<200;++j)
  i = midstr(str1,str2);
 timer(1);
 printf("midstr = %d\n",i);
 timer(0);
 for(j=0;j<200;++j)
  i = instr(1,str1,str2);
 timer(1);
 printf("instr = %d\n",i);
 }
}
```

Notes:

- o The first use for midstr() is to replicate the two-argument INSTR(A$,B$) found in Microsoft's MBASIC programming language.

- o This function is two to six times faster than instr(). Use it where speed counts, like in editor searches.

- o A portable version of midstr() appears below, in case you need to port the code to a different target. The portability causes the speed to be lost, but it is useful for bringing up a program for the first time. Fast implementations should not call strlen() or strncmp(). Assembler is mandatory even on a 10mhz 68000 CPU.

```
int midstr(s,t)
char *s,*t;
{
int i,j;
        i = strlen(t);
        j = 0;
        while(*s) {
             ++j;
             if(*s == *t && strncmp(s,t,i) == 0) return j;
             ++s;
        }
        return 0;
}
```

The min Function

==

Purpose:

Computes the minimum value of two integer arguments.

Function Header:

```
int min(x,y)
int x,y;                    Integers to compare.
```

Returns:

The smaller of x and y as signed integers, i.e., min(-1,-2) = -2 and min(1000,2000) = 1000.

Example: Print the minimum value of two numbers a,b entered at the console.

```
#include <stdio.h>
main()
{
char s[129];
int a,b;

     printf("Enter number a: ");
     gets(s); a = atoi(s);
     printf("Enter number b: ");
     gets(s); b = atoi(s);
     printf("min(%d,%d) = %d\n",a,b,min(a,b));
}
```

Notes:

- o The min() function used here is an honest function that works only on integers. It fails on long integers and floats.
- o For long integers, use min = (a < b ? a : b).
- o For floats, use min = (a < b ? a : b).
- o Most libraries assume the min() function is a macro. Look in the STDIO.H header file or in MATH.H. Such functions definitely have side effects.

The mktemp Function

==

Purpose:

Create new contents for the last 6 characters of a null-delimited string. The string can be used as a filename (up to 8 chars, no extension).

Function Header:

```
     char *mktemp(tmp)
     char *tmp;           Address of the template string to be altered.
                          The argument should be a NULL-terminated string
                          which ends in the six special characters
                          XXXXXX.
```

Returns:

Its argument, the string base address. Generally, the return value is not useful.

Example: Open a temporary file, write one byte, close it and delete the file from the directory.

```
        VOID testtmp()
        {
        FILE *fopen();
        FILE *fp;
        char *p,*mktemp();
        char name[20];

            strcpy(name,"@@XXXXXX");
            mktemp(name);
            strcat(name,".TMP");
            if(fp = fopen(name,"r")) {
                    puts("TMP file already in use");
                    exit();
            }
            if(fp = fopen(name,"w")) {
                putc('X',fp);
                fclose(fp);
            }
        }
```

Notes:

o It is up to the caller to supply a string as requested.

o The filename is derived by writing ascii digits over the XXXXXX portion of the string. Two leading characters are optional. An extension can be added using strcat().

The modf Function

==

Purpose:

Splits float x = f + n, where n is an integer and float f satisfies fabs(f) < 1.0.

Function Header:

```
float modf(x,nptr)
float x;                Float to split.
int *nptr;              Where to put the exponent.
```

Returns:

```
f              The fraction, a float remainder.
n              via *nptr = n;
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,f;
int n;
float atof();
float modf();
     printf("Enter float x: "); gets(s); x = atof(s);
     f = modf(x,&n);
     printf("modf(%f,nptr) = %f, *nptr = %d\n",x,f,n);
}
```

Notes:

o No error codes returned. No need to code for errno.

o No overflow check.

o No auto conversion to FLOAT.

The moveMEM Function

Purpose:

Copies one region of memory to another without ripple.

Function Header:

```
char *moveMEM(dest,source,n)
char *dest;                     Destination address.
char *source;                   Source address.
unsigned n;                     Number of bytes to fill.
```

Returns:

```
dest+n                          Next location after move.
```

Example: Copy a buffer to another location.

```
#include <stdio.h>
main()
{
char s[100];
char t[100];
     fillchr(s,'A',100);
     moveMEM(t,s,100);
}
```

Notes:

- o Usage of this function differs among libraries. Often called movmem() with arguments in a different order.

- o The order of moveMEM arguments is the same as strncpy() and strncat(), which is in turn the same order as fillchr(). This consistency evidently escaped the authors of other libraries.

- o No ripple means that the buffers being moved may be overlapping. There is no danger in using this function, whereas strncpy() may have problems.

- o In this library, auto-switching to a Z80 block move is made on capable CPU's. If the Z80 CPU test fails, then a standard 8080 block move is done.

The movmem Function

Purpose:

Non-ripple memory move. BDS-C, CI-86, WIZARD and LATTICE standard.

Function Header:

```
VOID movmem(source,dest,n)
char *source;                  Source address.
char *dest;                    Destination address.
int n;                         Number of bytes to move.
```

Returns:

Nothing.

Example: Copy a file control block and print it.

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

main()
{
char.fcb[36];
extern char *IOfcb[];
int fd,i;
FILE *fp,*fopen();

     fp = fopen("TMP","w");
     if(fp) {
          fd = fileno(fp);
          movmem(IOfcb[fd],fcb,36);
          fclose(fp); unlink("TMP");
          for(i=0;i<36;++i) printf("fcb[%d] = %u\n",i,fcb[i]);
     }
}
```

Notes:

- o This function calls the main library function called moveMEM. The argument order of moveMEM() is the same as strncpy(). The argument order of movmem() has the destination and source reversed from strncpy() and strncat().

- o A portable version of movmem exists, but it is slow:

```
movmem(s,d,n)
char *s,*d; int n;
{
int m;
   if(d<s) {
        while(n--) *d++ = *s++;
   }
   else {
        d += n; s += n;
        while(n--) *--d = *--s;
   }
}
```

- o The above source is the same as moveMEM(d,s,n). Both functions are generally optimized. On an 8088/8086 machine, the string primitives of the CPU are used. On a Z80 machine, the CPU block memory moves are used.

The numsort Function

==

Purpose:

Distribution sort for numbers in High-Speed Assembler. Implements Donald Knuth's MathSort algorithm for 16-bit numbers.

Function Header:

```
int numsort(n,table,scrap)
int n;                             Number of integers to sort.
int table[];                       Array of integer data. Minimum
                                   size of the table is 2n bytes.
int scrap[];                       Array, scratch space, 2n+512
                                   bytes minimum.
```

Returns:

```
0         Success.
```

Example: Sort a list of integers.

```
#include <stdio.h>
main()
{
int i;
static
int data[] = {
     9324,2352,3243,6556,5554,
     4445,4544,5421,4221,4343
};
auto int scrap[10+256];
   if(numsort(10,data,scrap) == 0)
     for(i=0;i<10;++i) printf("%u\n",data[i]);
}
```

Sample output:

```
2352
3243
4221
4343
4445
4544
5421
5554
6556
9324
```

Notes:

o The order of the arguments is essential. Actually, the integer n can be unsigned, as far as the assembler code is concerned. But the address space of an 8080 machine will not allow more than 32767 integers.

o Sorts 12000 numbers in 3 seconds at 4mhz on a Z80 machine using Digital Research CP/M 2.2.

o This routine can only be used on 16-bit data.

o The source code is written in 8080 assembler, but full C source appears in the source code comments. See the source archives.

o The use of an auto array can cause trouble. The total byte count for all auto arrays in a function cannot exceed 32767. If in doubt, then use sbrk() (or malloc()) followed by sfree() (or free()).

o The scrap array is used by numsort() but it does not return anything useful. The sorted data is returned in table[].

open, opena

The open, opena Functions

==

Purpose:

Ascii File open() by file descriptor.

Function Header:

```
#include <unix.h>
int  open(name,access)
char *name;            An ascii NULL-terminated filename string
int   access;          Access, an integer 0,1,2. ASCII text mode
                       only, as follows:

                       0       Read-only
                       1       Write-only
                       2       Update (read & write)
```

Returns:

-1	Open failed - not the same as fopen!
fd	Open worked, fd=file descriptor

WARNING: In UNIX.H the following definition is made:

```
#define open opena
```

Note that opena(), openb() are real functions and not macro definitions.

Example: Open a disk file by descriptor and write 128 bytes from a buffer.

```
#include <unix.h>
#include <stdio.h>
pump(buffer,file)
char *buffer,*file;
{
int fd;
     fd = open(file,1);        /* open for output */
     if(fd == -1) {
          puts("Bad open"); return;
     }
     write(fd,buffer,128);
     close(fd);
}
```

Notes:

- o Use of this function avoids seekend() and its corresponding overhead. Append access is not an option.

- o The CP/M devices "CON:", "RDR:", "LST:", "PUN:" can be opened by using this function. They are unbuffered files. Console input has a type-ahead buffer, which makes it a hybrid unbuffered file.

- o Binary mode is supported through openb().

- o Beware of the lack of portability of binary mode. When implemented, many libraries try to do so by extended coding of the access. The latter is the worst of choices since it transports invisible program bugs.

The openb Function

==

Purpose:

Binary File open() by file descriptor.

Function Header:

```
int  openb(name,access)
char *name;               An ascii NULL-terminated filename string
int   access;             Access, an integer 0,1,2. BINARY text mode
                          only, as follows:

                          0       Read-only
                          1       Write-only
                          2       Update (read & write)
```

Returns:

```
-1      Open failed - not the same as fopen!
fd      Open worked, fd=file descriptor
```

WARNING: In UNIX.H the following definition is made:

```
#define open opena
```

Note that opena(), openb() are real functions and not macro definitions.

Example: Open a disk file by descriptor and write 128 bytes from a buffer. The write will not translate CR/LF pairs.

```
#include <unix.h>
#include <stdio.h>
pump(buffer,file)
char *buffer,*file;
{
int fd;
     fd = open(file,1);        /* open for output */
     if(fd == -1) {
          puts("Bad open"); return;
     }
     write(fd,buffer,128);
     close(fd);
}
```

Notes:

- o Use of this function avoids seekend() and its corresponding overhead. Append access is not an option.
- o The CP/M devices "CON:", "RDR:", "LST:", "PUN:" can be opened by using this function. They are unbuffered files. Console input has a type-ahead buffer, which makes it a hybrid unbuffered file.
- o Binary support for the devices does not really exist. The console can be put into binary mode by using the function CHmode(). This is a CP/M kludge and not a portable feature.
- o Beware of the lack of portability of binary access. When implemented, many libraries try to do so by extended coding of the access. The latter is the worst of choices since it transports invisible program bugs.

The outport Function
==

Purpose:

Sends a value out a given CPU port.

Function Header:

```
int outport(port,value)
int port;                       Port number.
int value;                      Value to transmit.
```

Returns:

Nothing useful. Assumes the port is ready to receive a character. See the application notes and the following example.

Example: Send characters out the modem port.

```
#define MODEM       0330  /* Z90A computer, Z80 CPU, 255 ports */
#define OFFSET      0005  /* 8250 UART, line status register */
#define ISEMPTY     0040  /* test transmitter register empty */
#include <stdio.h>
main()
{
    fprintf(stderr,"Enter characters, end with ctrl-Z\n");
    while((x = getchar()) != EOF) {
         /* get char, wait for status, ship the byte */
         while((inport(MODEM+OFFSET)&ISEMPTY) == 0) ;
         outport(MODEM,x);
    }
}
```

Notes:

- o On most machines the status of the port must be interrogated in order to write to the data port. This is true regardless of the port buffering.

- o Disk I/O ports are often interrupt driven, so writing to the port will require set-up of the interrupt vector for return.

- o Clock and timer chips like the 8253 Intel make for good examples of port usage in C programs. See also Analog-to-Digital boards and parallel port interfaces. These appear on music boards and game interfaces. The signals are joysticks, mice, light pens, plotters, graphics tablets and the like.

- o Port operations are inherently machine-dependent. Most systems people don't care, however, because the problem being solved by port operations is usually special and one-time.

The Peekb, Peekw and Peekl Functions

Purpose:

Fetch a value from main memory, either 8, 16 or 32 bits.

Function Header:

```
char peekb(addr);
char *addr;                          Byte address.

unsigned peekw(addr);
char *addr;                          Int address.

long peekl(addr);
char *addr;                          Long integer address.
```

Returns:

(char)peekb()	Byte at address
(int)peekw()	Word at address
(long)peekl()	Long integer at address

Example: Access a long integer in three ways.

```
#define pfLONG 1
#include <stdio.h>
main()
{
char *addr;
long x;
char peekb(); unsigned peekw(); long peekl();

  x = 238823492;
  printf("x = 238823492\n");
  printf("byte=%c\n",peekb(&x));
  printf("word=%u\n",peekw(&x));
  printf("long=%ld\n",peekl(&x));
}
```

Notes:

o These functions are used when it is clumsy to use pointers.

o Pointers sometimes hide ideas. These functions can help to document otherwise obscure code.

o Beware of using peekl() without its proper declaration.

The perror Function

===

Purpose:

Writes a short message on the system console, describing the last system error, as detected by the variable errno. The message is derived from the string table sys_errlist[].

Function Header:

```
#include <errno.h>
char *sys_errlist[];        Error strings. See below.
int sys_nerr;               Number of error strings.
int  perror(s)
char *s;                    Prefix string to print prior to message.
```

Returns:

The value of errno before the call.

Example: Print the current error from variable errno.

```
#include <unix.h>
#include <math.h>
#define pfFLOAT 1
#include <stdio.h>
main()
{
float sin();
extern int errno;
     errno = 0;
     printf("sin(%f) = %f\n",5600.01,sin(5600.01));
     if(errno) perror("sin function error: ");
}
```

Notes:

o Not much of the library currently uses the errno variable. Mostly for the transcendental functions.

o Fixes to the library should go through this routine during debugging to insure that the proper messages get printed.

o The currently supported error message appear below:

```
char *sys_errlist[] = {
"Errno 0 detected",
"Permission denied",
"File not found",
"",     /* No such process */
"",     /* System call interrupted by signal */
"I/O error",
"No such I/O device",
"",     /* Too many arguments to exec */
"",     /* Wrong format for executable file */
"Bad file descriptor",
"",     /* No children */
"",     /* Cannot fork */
"Too little memory available",
"",     /* File access conflict with user rights */
"",     /* Bad memory address supplied */
"",     /* Wrong sort of device */
"",     /* Device already in use */
"The file already exists",
"",     /* You cannot link across devices */
"",     /* Silly access to this device */
"",     /* Directory name expected */
"",     /* Directory name not expected */
"",     /* Invalid argument */
"",     /* System out of file table space */
"",     /* Request for too many file descriptors */
"",     /* Not a teletype device */
"",     /* File currently in use */
"",     /* File became too large for system */
"Disk system out of space",
"",     /* Seek attempted on pipe */
"",     /* Read-only file system */
"",     /* Too many links to a file */
"",     /* Write attempt on broken pipe */
"Out of function domain",
"Result out of range"
};

int sys_nerr = sizeof(sys_errlist)/sizeof(sys_errlist[0]);
```

The Pokeb, Pokew and Pokel Functions

==

Purpose:

Store a value to main memory, either 8, 16 or 32 bits.

Function Header:

```
pokeb(addr,x);
char *addr;                    Address to store the byte
char x;                        8-bit data to be stored

pokew(addr,x);
char *addr;                    Address to store the word
unsigned x;                    16-bit data to be stored

pokel(addr,x);
char *addr;                    Address to store the long integer
long x;                        32-bit data to be stored
```

Returns:

Nothing useful

Example: Union manipulation using the poke functions.

```
#define pfLONG 1
#include <stdio.h>
main()
{
union { char c; int i; long x; } xx;

  printf("Answers should be: 65, 65000, 6500000\n");
  pokeb(&xx.c,(char)65);
  printf("character xx.c=%u\n",xx.c);
  pokew(&xx.i,(unsigned)65000);
  printf("integer xx.i=%u\n",xx.i);
  pokel(&xx.x,(long)6500000);
  printf("long xx.x=%ld\n",xx.x);
}
```

Notes:

o These functions are used when it is clumsy to use pointers.

o Pointers sometimes hide ideas. These functions can help to document otherwise obscure code.

o A typical application in systems is pokew(peekw(1)-16). While it can be done with pointers, the above is easier to debug.

The pow Function

==

Purpose:

Compute the power of real float values x,y i.e., computes x raised to power y.

Function Header:

float pow(x,y)	Standard function x^y.
float x,y;	Base = x, exponent = y.

Returns:

number	x and y in range
1.0	x nonzero and y = 0
INF	arguments give answer too large
	errno = ERANGE returned to flag error
-INF	x < 0 and y not an integer
-INF	x = 0 and y <= 0
	errno = EDOM returned to flag error
-pow(-x,y)	x < 0 and y = odd integer
pow(-x,y)	x < 0 and y = even integer

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,y;
extern int errno;
float atof();
float pow();
     errno = 0;
     printf("Enter float base x: "); gets(s); x = atof(s);
     printf("Enter float exponent y: "); gets(s); y = atof(s);
     printf("pow(%f,%f) = %f, errno = %d\n",x,y,pow(x,y),errno);
}
```

Notes:

o Table lookup. See C&H 6.2.34. Tables from C&H 1403.

o This function tends to be a bit slow.

o pow() uses the power of 10 function pow10(), which does limited error checking.

The pow10 Function

Purpose:

Compute power base 10 of real float value y i.e., computes 10 raised to power y.

Function Header:

float pow10(y)	Standard function 10^y.
float y;	Base = 10, exponent = y.

Returns:

number	Assume y in range.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float y;
extern int errno;
float atof();
float pow10();
     errno = 0;
     printf("Enter float exponent y: "); gets(s); y = atof(s);
     printf("pow10(%f) = %f, errno = %d\n",y,pow10(y),errno);
}
```

Notes:

- o This function does almost no error checking because the true front end is the pow() function. This part of the code is slow already, and any additional error checking seems not to be needed in a float-only library.

The printf Function

Purpose:

Performs output formatting for the standard output stream stdout. Uses a control string followed by appropriate arguments.

Function Header:

```
int printf(control,arg1,arg2,...)
char *control;                  Control string, see below.
arg1,arg2,...                   Appropriate arguments, see below.
```

The control string contains:

White space	Tabs, spaces, return, formfeed, vertical tab, newline, backspace
Escapes	Octal values prefixed by \, e.g., \007
Characters	Ordinary printing characters other than %
%%	Prints as %
%	Conversion flag

Following a conversion flag %, there will be an optional FLAG CHARACTER:

1. minus sign, signifies left-justification rather than right.
2. zero, means pad with zero rather than spaces.

Harbison & Steele recommend other modifiers, which are not supported:

3. plus sign, means always output + or - with numbers.
4. space, means always produce - or space fill.
5. #, means use a variant of the main conversion operation.

Following the flag character:

6. Field width, a sequence of decimal digits. Conversions are padded to this width, unless the field width is exceeded, in which case the whole field is printed and padding is ignored.

Following the field width:

7. An optional period and string of decimal digits, whose value is used to control the number of floating-point digits to the right of the decimal point. This called the Precision Field.

Following the precision field:

8. An optional LONG specifier, expressed as lowercase l, which is used where such a prefix makes sense. This is called the Long Prefix Field.

Following the long prefix field:

9. A conversion operation, expressed as a single character taken from the lowercase letters:

c	character
s	string
d	signed decimal base 10
u	unsigned decimal base 10
o	octal base 8
x	hexadecimal base 16
b	binary base 2
e	scientific format float, exponents
f	float format without exponents
g	smaller of e and f formats, strip zeros

The conversions E, G, X are not supported. The possible long conversions are:

ld	long signed decimal base 10
lo	long octal base 8
lx	long hexadecimal base 16
lb	long binary base 2
lu	long unsigned decimal base 10

Hexadecimal conversions use capital letters only. Binary conversion is provided as a debugging convenience. Do not expect it to be a portable feature of printf. All scientific exponents use lower case e rather than E.

Returns:

EOF	Error occurred for output
?	No standard exists for return otherwise. Probably should be the number of characters printed.

Under C/80, extra parentheses are required to capture the return value of printf, e.g., (printf(...)). But CP/M cannot tolerate I/O on a full disk, so the library gracefully warm boots, long before you could capture an EOF indication.

Example: Control of field width.

```
#define mathlib 1
#include <stdio.h>
#undef printf
#define printf prnt_1(),prnt_2          /* select portable printf */
main()
{
static char *s = "Hello World";
static char c = 'A';
  printf("%10s :%10s:\n",":%10s:",s);
  printf("%10s :%-10s:\n",":%-10s:",s);
  printf("%10s :%20s:\n",":%20s:",s);
  printf("%10s :%-20s:\n",":%-20s:",s);
  printf("%10s :%20.10s:\n",":%20.10s:",s);
  printf("%10s :%-20.10s:\n",":%-20.10s:",s);
  printf("%10s :%.10s:\n",":%.10s:",s);
  printf("%10s :%c:\n",":%c:",c);
  printf("%10s :%10c:\n",":%10c:",c);
  printf("%10s :%-10c:\n",":%-10c:",c);
  printf("%10s :%010c:\n",":%010c:",c);
  printf("%10s :%-010c:\n",":%-010c:",c);
  printf("%10s :%20s:\n",":%20s:","");
}
```

```
    :%10s: :Hello World:
   :%-10s: :Hello World:
    :%20s: :         Hello World:
   :%-20s: :Hello World         :
 :%20.10s: :         Hello World:
:%-20.10s: :Hello Worl          :
   :%.10s: :Hello Worl:
      :%c: :A:
    :%10c: :         A:
   :%-10c: :A         :
   :%010c: :000000000A:
  :%-010c: :A000000000:
    :%20s: :                    :
```

In this library, precision is the string length of the conversion regardless of the selected precision. As stated in K&R, precision takes effect only for float numbers and strings.

Example: Control of decimal padding character (0 or space).

```
#define mathlib 1
#include <stdio.h>
#undef printf
#define printf prnt_1(),prnt_2          /* select portable printf */
main()
{
static int x = 1001;
    printf("%10s :%10d:\n",":%10d:",x);
    printf("%10s :%-10d:\n",":%-10d:",x);
    printf("%10s :%010d:\n",":%010d:",x);
    printf("%10s :%10x:\n",":%10x:",x);
    printf("%10s :%-10x:\n",":%-10x:",x);
    printf("%10s :%010x:\n",":%010x:",x);
}
```

```
Output of the program:
:%10d: :      1001:
:%-10d: :1001      :
:%010d: :0000001001:
:%10x: :       3E9:
:%-10x: :3E9       :
:%010x: :00000003E9:
```

Example: Control of floating point precision.

```
#define pfFLOAT 1
#include <stdio.h>
main()
{
static float f = 45.3483;
    printf("%10s :%8.6f:\n",":%8.6f:",f);
    printf("%10s :%-8.6f:\n",":%-8.6f:",f);
    printf("%10s :%08.6f:\n",":%08.6f:",f);
    printf("%10s :%20.10f:\n",":%20.10f:",f);
    printf("%10s :%-20.10f:\n",":%-20.10f:",f);
    printf("%10s :%020.10f:\n",":%020.10f:",f);
    printf("%10s :%8.6e:\n",":%8.6e:",f);
    printf("%10s :%-8.6e:\n",":%-8.6e:",f);
    printf("%10s :%08.6e:\n",":%08.6e:",f);
    printf("%10s :%20.10e:\n",":%20.10e:",f);
    printf("%10s :%-20.10e:\n",":%-20.10e:",f);
    printf("%10s :%020.10e:\n",":%020.10e:",f);
    printf("%10s :%8.6g:\n",":%8.6g:",f);
    printf("%10s :%-8.6g:\n",":%-8.6g:",f);
    printf("%10s :%08.6g:\n",":%08.6g:",f);
    printf("%10s :%20.10g:\n",":%20.10g:",f);
    printf("%10s :%-20.10g:\n",":%-20.10g:",f);
    printf("%10s :%020.10g:\n",":%020.10g:",f);
}
```

Output of the program:

```
   :%8.6f: :45.348300:
  :%-8.6f: :45.348300:
  :%08.6f: :45.348300:
 :%20.10f: :       45.3483000000:
:%-20.10f: :45.3483000000       :
:%020.10f: :000000045.3483000000:
   :%8.6e: :4.534830e+01:
  :%-8.6e: :4.534830e+01:
  :%08.6e: :4.534830e+01:
 :%20.10e: :    4.5348300000e+01:
:%-20.10e: :4.5348300000e+01    :
:%020.10e: :00004.5348300000e+01:
   :%8.6g: : 45.3483:
  :%-8.6g: :45.3483 :
  :%08.6g: :045.3483:
 :%20.10g: :             45.3483:
:%-20.10g: :45.3483             :
:%020.10g: :000000000000045.3483:
```

Notes:

- o Printf is not recursive. This means that you cannot have the result of a printf, fprintf or sprintf function call in the argument list for printf.

- o The recursion loss is in the #define kludge for multiple arguments. The stack location is stored in a static variable and not on the stack, so repeated calls over-write the stack location.

- o The small printf in the main library does have long or float support and does not pretend to be K&R standard, although it comes very close. Its advantage is speed, and to some extent, recursion.

- o Binary conversion %b is not supported by very many libraries, however it is an invaluable debugging tool, especially in view of the lack of bit fields in C/80. Try to restrain yourself and use it only in debug code.

- o The fat for printf comes largely from support code for long and float data types. See the examples above for switching code to turn on just what you need in your application.

The putc Function

==

Purpose:

Writes a character to an open stream.

Function Header:

```
int putc(x,fp)
int x;                      Character to output
FILE *fp;                   Open stream pointer
```

Returns:

```
-1             If end of media was reached. We define EOF to be -1.
               End of file on output is either an error or end of
               media (out of disk space).
 c             Character to be written (x), on success.
```

Example: The following writes characters to a file entered on the command line until ctrl-Z is entered. The explicit use of fclose() is required to enter the file information into the disk directory and flush any orphan record to disk.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x;
FILE *fp,*fopen();
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"w")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((x = getchar()) != EOF) {
          putc(x,fp);
     }
     fclose(fp);
}
```

Notes:

o This function is not a macro in the present library. Expect it to be a macro in most C libraries. It has no side effects.

o Under CP/M, the BIOS will complain on a disk write error, which is the only possibility besides end of file. In this case, the system will kill the running program unless you insist upon continuing with the error.

o A return of -1 under C/80 means that end of media was encountered. Otherwise, expect the character to be returned. Output to devices PUN: and LST: are expected to be subject to CR/LF translation unless done in binary mode. See putcbinary() and writeb().

o Beware of mixing file descriptors and stream pointers. While C/80 will buy it, other systems won't. The connection is fd = fileno(fp), where fd is an integer and fp is a stream pointer.

o C/80 will not allow an output to a full disk. Generally, putc() fails because of no disk space, and you will be bounced out to the system.

The putcbinary Function

Purpose:

Writes a character to an open stream in binary mode. Not a K&R function. Recognizes CON:, LST:, PUN: devices.

Function Header:

```
int putcbinary(x,fp)
int x;                        Character to output
FILE *fp;                     Open stream pointer
```

Returns:

-1	If end of file was reached. We define EOF to be -1. For output, EOF means end of media.
c	Character to be written (x), on success.

Example: The following writes characters to stream stdout until the user types ctrl-Z. All characters are written in binary mode.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x;
     while((x = getchar()) != EOF) {
          putcbinary(x,stdout);
     }
}
```

Notes:

- o A return of -1 under C/80 means that end of media was encountered. Otherwise, expect the character to be returned. Output to devices CON:, PUN: and LST: are expected to be subject to CR/LF translation.

- o This function is a library internal, documented for your convenience. Do not expect it to be present on other systems.

- o Since RDR: is a read-only device, putcbinary() will come back with an error for that stream. Use PUN:, also called AXO:.

- o It is more portable to use putcbinary() than to constantly use the non-portable bios() features. Most target machines will support a function like putcbinary(), but the details of how to access binary output mode will vary across machines.

putchar

The putchar Function

Purpose:

Writes a character to the standard output stream stdout.

Function Header:

```
int putchar()
```

Returns:

c	Character output to stream stdout, on success.
-1	If an output error occurred, such as no disk space.

Example: The following writes characters to the console until ctrl-Z is encountered.

```
#include <stdio.h>
main()
{
int x;
     while((x = getchar()) != EOF) {
          putchar(x);
     }
}
```

Notes:

- o Expect putchar to be a macro in most C libraries. It has no side effects. In the present library it is an honest function.
- o Under CP/M, the BIOS will complain on a disk write error, which is the only possibility.
- o A return of -1 under C/80 means that an error was encountered.
- o putchar() writes to stream stdout, which may in fact be a file due to re-direction.
- o If stream stdout is the console CON: or the punch PUN: (modem), then CR/LF translation is assumed. To disable this feature, use writeb() or putcbinary(). Under Bell Labs Unix systems, a newline character is used instead of the CP/M CR/LF pair, hence the need to convert will not exist on such systems.

The putl Function

==

Purpose:

Writes a 32-bit long integer to the stream, in Intel Reverse Format.

Function Header:

```
long putl(x,fp)
long x;                  32-bit integer to output to file
FILE *fp;                Stream pointer, open stream
```

Returns:

```
 x       32-bit word, on success
-1       Error or end of file
```

Example: Write out long integers to a file.

```
VOID dumplongs(fp,fi,n)
FILE *fp;                         Open output stream
FILE *fi;                         Open input stream
int n;                            Number of elements
{
long x,putl(),getl();
  while(n) {
        if((x = getl(fi)) == EOF) {
            if(feof(fi) == EOF) break;
        }
        if(putl(x,fp) == EOF) {
            if(x != EOF) break;
        }
        --n;
  }
}
```

Notes:

- o To detect end of media, check putl(x,fp) != x && x != -1.
- o Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. Other libraries may use putc() to write putl(), thereby allowing translation of characters data corruption.
- o Files opened for ASCII text mode usually cannot output a long integer using putc() or fputc() due to the danger of CR/LF translation. The latter is turned off during calls to putl() by using a special library function **putcbinary()**.

The puts Function

Purpose:

Writes characters to open stream stdout and appends a newline.

Function Header:

int puts(s)	
char *s;	Base address of the string to be output.

Returns:

EOF	Error occurred.
c	Success, last character c which was output.

Example: A version of ECHO.C.

```
#include <stdio.h>
main()
{
char *gets();
char s[129];
  while(TRUE) {
    if(stdin == (FILE *)0)
    fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
    if(gets(s) == (char *)0) break;
    puts(s);
  }
}
```

Notes:

- o An error for puts() is usually a disk space error, therefore the library will dump the user to the system. CP/M will not tolerate I/O to a full disk.

- o puts() differs from fputs() in that it appends a newline after the string is output. This newline is in addition to any newline that might be in the string.

The putw Function

Purpose:

Writes a 16-bit word to the stream, in Intel Reverse Format.

Function Header:

```
     unsigned putw(n,fp)
     unsigned n;                  16-bit unsigned integer to output
     FILE *fp;                    Stream pointer for the output file
```

Returns:

```
     n       16-bit word, on success.
    -1       Error (write-protected) or end of media.
```

Example: Write words to a file and check for end of media.

```
     pumpword(fp,x)
     FILE *fp;
     unsigned x;
     {
          if(putw(x,fp) == EOF) {
             if(x != EOF) return -1;
          }
          return 0;
     }
```

Notes:

- o Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. It may be possible that another target machine uses putc() to write putw(), in which case character translation can occur during output.

- o Files opened for ASCII text mode usually cannot output an integer using putc() or fputc() due to the danger of CR/LF translation. The latter is turned off during calls to putw() because of the explicit use of the special library function putcbinary().

The qsort Function

==

Purpose:

Quicksort with center pivot, stack control, and easy-to-change comparison method.

This version sorts fixed-length data items. It is ideal for integers, longs, floats and packed string data without delimiters.

Function Header:

```
int  qsort(base,n,s,CMP)
char *base;                    Base address of the raw string data
int  n;                        Number of blocks to sort
int  s;                        Number of bytes in each block
int  (*CMP)();                 Compare routine for two block pointers
                               p,q that returns an integer with the
                               same rules used by Unix strcmp(p,q):

                                  = 0     Blocks p,q are equal
                                  < 0     p < q
                                  > 0     p > q
```

Beware of using ordinary strcmp() - it requires a NULL at the end of each string.

Returns:

0   Always

Example: Sort an array of integers.

```
#include <unix.h>
#include <stdio.h>
int mycmp(p,q) int *p,*q; { return (*p - *q);}
int q[10] = {12,1,3,-2,16,7,9,34,-90,10};
int p[10] = {12,1,3,-2,16,7,9,34,-90,10};
main()
{
int i;
        qsort(p,10,2,mycmp);
        for(i=0;i<10;++i) printf("%d. %d, %d\n",i,p[i],q[i]);
}
```

Output from the above sample program

```
0. -90, 12
1. -2, 1
2. 1, 3
3. 3, -2
4. 7, 16
5. 9, 7
6. 10, 9
7. 12, 34
8. 16, -90
9. 34, 10
```

Notes:

o Qsort() can sort raw integers, longs, floats or strings. However, the string sort is not efficient.

o Use quick() to sort string pointer arrays.

o Use cmpi(), cmpl(), cmpf(), cmps() to compare integers, longs, floats and strings, respectively.

References:

BYTE Oct-84, p 369	The Unix Book, p 200
By William M. Raike	by Banahan & Rutter
B*Y*T*E J*A*P*A*N	John Wiley & Sons, NY (1983)

quick

The quicksort Function

Purpose:

Quicksort with center pivot, stack control, and easy-to-change comparison method.

Function Header:

```
quick(lo,hi,base,CMP)
 int  lo;                First array subscript
 int  hi;                Last array subscript
char *base[];            Base address of pointer array
 int  (*CMP)();          Compare routine for two strings p,q
                         that returns an integer with the
                         same rules used by Unix strcmp(p,q):

                            = 0     strings p,q are equal
                            < 0     p < q
                            > 0     p > q
```

Returns:

```
0       Always
```

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
        quick(0,9,p,strcmp);
        for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a                Numerals are less than letters
1. 1, C                in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3                UPPERCASE is less than lowercase
8. a, 8                in the ASCII standard
9. d, 5
```

Notes:

- o You can most often use strcmp() as the argument for cmp().
- o It is necessary to declare the function cmp() before the call to quick(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.
- o To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example below.
- o This source file is very portable. Use it on any system with minimal C compiler.

Reference:

BYTE Oct-84, p 369
By William M. Raike
B*Y*T*E J*A*P*A*N

The rad Function

==

Purpose:

Changes float x degrees to radians.

Function Header:

```
float rad(x)          Radian value of argument x.
float x;              Float value in degrees.
```

Returns:

angle in radians (a FLOAT)

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sin(),rad();
     printf("Enter float degrees x: "); gets(s); x = atof(s);
     printf("sin(rad(%f)) = %f, rad(%f) = %f, errno = %d\n",
            x,sin(rad(x)),x,rad(x),errno);
}
```

Notes:

o Method used is angle = (PI/180.0)*x.

o No error codes returned. No need to code for errno.

o No overflow check.

o No auto conversion to FLOAT.

The rand and srand Functions
==

Purpose:

Rand() generates pseudo-random numbers. Default seed is 2168. To get other seeds you must call srand().

Function Header:

```
     unsigned rand()          Returns next random number in sequence.

     VOID srand(seed)         Seeds the random number generator.
     int seed;                Seed must be a 16-bit integer.
```

Returns:

Unsigned integer in the range 0 to 65535.

Example: Seed the random number generator from the Z100 2ms clock, then print 10 random numbers.

```
     #include <unix.h>
     #include <stdio.h>
     main(argc,argv) int argc; char **argv;
     {
     int i,peekw(),rand();

          srand(peekw(11));
          for(i=0;i<10;++i) printf("%u\n",rand());
     }
```

Notes:

- o Srand(seed) initializes the pseudorandom number generator with the value of seed. The same seed produces the same series of numbers from function rand().

- o The default value of 2168 is used in case you fail to seed the generator. This makes for rapid testing of programs. The seeding method is left unresolved until it becomes important.

- o Older versions of the library included a function called randl(). It still exists, for those cases when you want to have a large number of random number generators running simultaneously. See the source archives.

The rdDISK Function

==

Purpose:

Direct-disk read function for CP/M 2.2.

Function Header:

```
rdDISK(track,record,drive,buffer)

int track;                Physical track to read, 0 = first.
int record;               Record to read from track, 1 = first.
int drive;                Drive number. Use 0 for A:, 1 for B:, etc.
char buffer[128];         Location to copy disk data.
```

Returns:

Buffer filled on success.
Error code for a disk read is returned. Should be 0 for success.

Example: Read track 0 sector 1, the Zenith label sector.

```
#include <stdio.h>
main()
{
int i; char buf[128];
#define TRK 0
#define SEC 1
#define DSK 0
     printf("rdDISK() = %d\n", i = rdDISK(TRK,SEC,DSK,buf));
     for(i=0;i<128;++i) printf("%02x\n",buf[i]&255);
}
```

Notes:

- o This was used to code the CROSS-CHECK source for the Z90. See Dr.Dobbs Journal, September, 1983. It is also useful for writing your own disk patcher or disk scanner.

- o Note that rdDISK() is coded in assembly language and is highly non-portable. The function rdDISK() can be written in terms of the function bios(). This is recommended for portability, especially to CP/M- 68K and Unix. Here's the portable source:

```
bios(12,buffer);              Under CP/M-68K the second
bios(9,drive);                argument of bios() has a
bios(10,track);               cast of (long).
bios(11,record);              Under CP/M-80 2.2, the cast
return bios(13,0);            is (int).
```

- o Use bios(13,1) to do an immediate read that gets around any LRU buffering scheme. Note that rdDISK() uses bios(13,0).

read_ - C/80 Standard read

The read_ Function

Purpose:

Reads a block of characters off the disk in binary mode. The block must be a multiple of 128 bytes. Adheres to the Software Toolworks standard for read(). Use UNIX.H to obtain the Bell Labs Unix standard.

Function Header:

```
#include <stdio.h>
unsigned read_(fp,buffer,count)
FILE *fp;                          Open stream pointer.
char *buffer;                      Buffer for the read operation.
unsigned count;                    Number of bytes to read.
                                   Must be a multiple of 128.
```

Returns:

The number of bytes actually read.
The number 0 is returned if EOF is reached.

Example: Read 2048 bytes from a disk file.

```
#include <stdio.h>
get2048(fp,buf)
FILE *fp;
char buf[2048];
{
unsigned x,read_();
     x = read_(fp,buf,2048);
     if(x != 2048) puts("Read less than 2048");
     if(x == 0) puts("Probably EOF");
}
```

read_ - C/80 Standard read

Notes:

o Streams with descriptors 252,253,254,255 are devices. The C/80 read_() function cannot use these device descriptors. Use getc() for safety.

o Some systems return -1 on error. Your source code should always check read_() <= 0 rather than read_() == 0.

o A read failure is usually due to end of file. Since the C/80 read_() function operates in binary mode only, the ctrl-Z marker at the end of an ASCII file is not considered.

o The read_() function under C/80 operates in quanta of 128 bytes. You cannot read 1 character. See UNIX.H and the Unix-style read() and fread() functions for an alternative.

o This read_ function uses CP/M function 33 with manual advance of the record number. The initial record number is saved in the system variable IOsect[fp].

o The symbol **read_** is the same as **read** in CLIB.REL. Portable source code should use read_ instead of read. Old C/80 code should compile without changes. Unix code should use UNIX.H.

The read, reada Functions

===

Purpose:

Transfers bytes from a non-stream file to main memory. Move n bytes per call to a pre-assigned buffer. The transfer suffers from cr/lf translation if the file was opened in Ascii mode (the usual case).

Function Header:

```
int  reada(fd,buffer,n)
int fd;                   File descriptor, fd=fileno(fp) where
                          fp is the stream pointer.
char *buffer;             Pointer to base of memory storage
                          large enough to accept byte transfer.
int n;                    Number of bytes to transfer, 0...32767.
```

Returns:

```
 m      The number of bytes actually transferred, an
        integer quantity 0...32767.
-1      Error
```

WARNING: In UNIX.H appears the definition

```
                        #define read reada
```

Example: Read 1 byte from the standard input.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x;
     read(fileno(stdin),&x,1);
     fprintf(stderr,"Byte read = %d ASCII\n",x);
}
```

Notes:

- o Useful for reading in Ascii data from the current file pointer.
- o Recognizes devices "CON:", "RDR:".
- o The primitive read() is used implicitly, but all Unix redirection is in force, because of the explicit use of getc().
- o The UNIX.H definition of read() is reada(). It does correct I/O but suffers a slowdown with small buffer sizes.

The readb Function

==

Purpose:

Transfers bytes from a non-stream file to main memory. Move n bytes per call to a pre-assigned buffer. All byte transfers are in binary mode- no cr/lf translation.

Function Header:

```
int  readb(fd,buffer,n)
int fd;                   File descriptor, fd=fileno(fp) where
                          fp is the stream pointer.
char *buffer;             Pointer to base of memory storage
                          large enough to accept byte transfer.
int n;                    Number of bytes to transfer, 0...32767.
```

Returns:

m	The number of bytes actually transferred, an integer quantity 0...32767.
-1	Error

WARNING: In UNIX.H appears the definition

```
#define read reada
```

Example: Read one byte, binary mode, from a file entered on the command line. If it works, then print the byte.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
FILE *fp,*fopenb();
int x;
     if(argc > 1) {
       fp = fopenb(argv[1],"r");
       if(fp) {
          read(fileno(fp),&x,1);
          fprintf(stderr,"Byte read = %u\n",x);
     }
}
```

Notes:

- o Useful for reading in binary data. All I/O is stream type. This function uses the library primitive getcbinary().
- o The UNIX.H definition of read() is reada(), which does Ascii I/O correctly on files and devices.
- o To get no-echo input, the best procedure is to write a special function like getbyte() that will at least cause the ported code to encapsule the problem. This feature is available on all systems but is indeed sometimes difficult to interface.
- o Text editing features, ctrl-B processing and end of file detection other than end of record are all disabled under readb() when using the console device.

The realloc Function

Purpose:

Re-allocates contiguous memory located between the end of the program and the stack, copying the contents to the new region as required. Applies only to regions assigned by malloc().

Function Header:

char *realloc(ptr,s)	
char *ptr;	Pointer to area already assigned by malloc().
unsigned s;	New size of area.

The contents at ptr are preserved during the reallocation. If the new size s is smaller than the old size, then the extra contents are lost. Otherwise, new space is added on the end (not initialized), which will result in a memory move operation (a bit slow sometimes) which copies the contents.

Returns:

The base address of the area, on success.
0 if the request fails.

Example: Re-allocate a buffer area.

```
#include <unix.h>
#include <stdio.h>
main()
{
char *p,*malloc(),*realloc();

     p = malloc(2048);
     if(p == (char *)0) exit();
     printf("malloc(2048) = %x\n",p);
     for(i=0;i<10;++i) strcat(p,"This is a test of realloc\n");
     if(p = realloc(p,260)) {
       printf("realloc base = %x\n",p);
       for(i=0;i<260;++i) putchar(*p++);
     }
}
```

Notes:

- o Uses malloc() to get raw system memory.
- o You can't ask for more than 65531 bytes under CP/M-80.
- o Sfree() doesn't know about realloc().
- o Brk() doesn't know about realloc().

Structure: The structure used by malloc(), which is 4 bytes in length, precedes each block assigned by malloc(). Here is the definition:

```
struct block {
        struct block
             *nxtblk;
        unsigned
                 siz;
};
```

The rename Function

===

Purpose:

Change the name of an existing disk file.

Function Header:

```
int rename(oldname,newname)
char *oldname;                       Old file name.
char *newname;                       New file name.
```

Returns:

0	No error.
-1	Error, rename operation failed.

Example: Rename a disk file.

```
#include <stdio.h>
main()
{
int rename();
char *findFIRST();
char s[129],t[129];
        printf("File name to change: ");
        gets(s);
        if(findFIRST('\0',s) == (char *)-1) {
               puts("File not found"); exit();
        }
        printf("New name: ");
        gets(t);
        if(rename(s,t) == -1) puts("Rename failed");
}
```

Notes:

- o This function is found in most libraries but seems to be non-standard. Amazingly enough, the argument order is consistent among the major compilers.
- o The related function unlink() which deletes a disk file is a Bell Labs standard function.

The reset Function

==

Purpose:

Resets the disk system under CP/M.

Function Header:

```
VOID reset()
```

Returns:

Nothing useful. Drive A: is logged-in, then the default drive. All other drives are left in an unlogged state.

Example: Let the user change disks.

```
#include <stdio.h>
dskreset(drive)
char *drive;
{
char s[129];
  printf("Insert a new disk in %s and hit RETURN: ",drive);
  gets(s);
  reset();
}
```

Notes:

o Reset() is not a standard function. It is handy for writing interactive CP/M software.

o To simulate reset() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>
reset()
{
   return bdos(13,0);
}

#include <stdio.h>
reset()
{
   return ccBDOS(13);
}
```

The reverse Function

==

Purpose:

Reverses a string in place, e.g., "abcd" changed to "dcba".

Function Header:

```
char *reverse(str)
char *str;                  Source string.
```

Returns:

```
str                         Same base address, string reversed.
```

Example: Reverse a string of digits obtained by division.

```
#include <stdio.h>
main()
{
char s[129];
int i,n;
     printf("Enter a number: ");
     gets(s);
     n = atoi(s);
     i = 0;
     while(n>0) {
          s[i] = '0' + (n%10);
          ++i; n = n/10;
     }
     s[i] = '\0';
     printf("Number n before reversal: %s\n",s);
     reverse(s);
     printf("Number n after reversal: %s\n",s);
}
```

Notes:

- o This function is described in K & R.
- o Other uses of reverse() include swapping the string order in order to use strcpy() to copy a string into a circular buffer (in reverse order).
- o An interesting exercise is to employ getline() to get a string, then reverse it and use ungetc() to put it back into the console buffer. Follow all this by getline() to see if its the same.

The rewind Function

Purpose:

Positions an open stream to the beginning of the file.

Function Header:

```
#include <stdio.h>        VOID and FILE defined herein.
VOID rewind(fp)
FILE *fp;                 Open stream pointer.
```

Returns:

Nothing.

Example: Rewind a file before changing the file buffer location. Access the file with a new in-memory buffer of user design.

```
#define CPMEOF 26
#define bufszl 2048
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char buf[bufszl];
FILE *fp,*fopen();
int i;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("File not found"); exit(0);
     }
     if(getc(fp) == EOF) {
          puts("File empty"); exit(0);
     }
     rewind(fp);
     setbuf(fp,buf);
     getc(fp);
     for(i=0;i<bufszl;++i) {
          if(buf[i] == CPMEOF) break;
          putchar(buf[i]);
     }
}
```

Notes:

- o Rewind() has less overhead than fseek() or the older seek(). It does very little other than reset data areas in the FCB. No disk action takes place until I/O occurs.

- o Setbuf() is supposed to include a call to rewind(). The above code is especially careful, because ported code may end up with a call to a setbuf() that does not call rewind.

- o The size of a file buffer is fixed at compile time with most compilers. This library is unusual, in that the buffer size can be changed on the fly. However, it should always be preceded by a call to rewind(), to insure past data is on the disk and I/O starts anew.

- o The companion function seekend() is useful to position the file pointer to the end of the file.

- o While rewind is usually available on target systems, it generally calls fseek(). Note that seekend() is not expected to be on other systems. Both can easily be defined as a **code macro** in terms of fseek:

```
#define rewind(fp)  fseek(fp,0L,0)
#define seekend(fp) fseek(fp,0L,2)
```

The rindex Function

==

Purpose:

Finds the character position of a byte inside a string, starting from the end of the string.

Function Header:

```
     char *rindex(str,c)
     char *str;                  Source string.
     char c;                     Character to locate.
```

Returns:

&str[i]	Where c == str[i], on success, where i =strlen(str) is decremented to 0, and the first match is reported.
(char *)0	Failure.

Example: Find the optional extension in a file name and print.

```
     #include <stdio.h>
     main()
     {
     char *rindex();
     char s[129],t[129],*p;
          printf("Enter a file name: ");
          gets(s);
          cvupper(s);
          if(p = rindex(s,'.')) strcpy(t,p+1);
          else strcpy(t,"");
          printf("File name extension: %s\n",t);
     }
```

Notes:

- o Usage of this function differs among libraries. The Unix V7 standard, according to Bourne, is the above.
- o This function should be capable of finding the terminating null in a C string.
- o Rindex() is the same function as **strrchr()**. See Harbison & Steele.

The run Function

Purpose:

Runs a CP/M 2.2 command line directly from a C program. The current program is lost and replaced by the new one. All CP/M command line arguments are sent to the new program.

Function Header:

```
VOID run(cmd)
char *cmd;               Program name (.COM assumed) plus the
                         command line tail.
```

Returns:

Nothing. The routine run() returns to the caller only in case the file is not found. Otherwise, the COM file is loaded and the the command line is passed.

Example: Run MBASIC.COM from a C program with a given command line.

```
#include <stdio.h>
main()
{
     run("MBASIC DUNGEON");
     puts("MBASIC not found");
}
```

Notes:

o Multiple arguments may be used. The command line limit of 128 characters must be obeyed, however. The .COM extension should not be used - it is assumed.

o All characters are acceptable, however uppercase translation is performed to simulate CP/M handling of command lines.

o The string may not contain a carriage return or linefeed.

o The string tail is decoded into the two CP/M default buffers and in addition it is copied to the CP/M default buffer at 0x80. The only real difference that CP/M might see is the lack of a command tail at the CCP. This problem has no easy solution, since the CCP is overlayed by the running program.

The sbinary Function

==

Purpose:

Binary search of a sorted string array to match a key string.

Function Header:

```
int sbinary(x,v,n)
char *x;                     Key string to find in the array v[].
char *v[];                   Sorted string array for lookup.
int n;                       Dimension of the array v[].
```

Returns:

```
-1       Failure. The test (strcmp(v[k],x)==0) failed for 0 <= k < n.
 k       Success. A value of TRUE was found for (strcmp(v[k],x)==0).
```

Example: Look up a terminal in a sorted table.

```
static char *v[] = {
  "ADDS", "H19", "QUME", "TELERAY", "TELEVIDEO", "VISUAL200",
  "VISUAL500", "VT52", "WYSE", "Z29"
};
#include <stdio.h>
main()
{
int k;
char x[129];
     printf("Enter terminal name: ");
     gets(x); cvupper(x);
     k = sbinary(x,v,10);
     if(k == -1) puts("Not found");
     else
     printf("Found at index %d: %s\n",k,v[k]);
}
```

Notes:

- o The assumption that the array v[] is SORTED is essential. It will probably hang the machine if the array is not sorted.
- o A normal application uses sorted data entered in the program by hand as initializers for array v[].
- o To obtain a sorted array on the fly use quick() or shell().

The sbrk Function

==

Purpose:

Obtains contiguous memory from the system and sets the upper bound of the program+data area.

Function Header:

```
char *sbrk(n)
int n;                  Amount of memory requested. The maximum value
                        of n is 32767. See notes below.
```

Returns:

```
addr                    on success, addr = base address of the
                        contiguous area of memory n bytes in length.
(char *)-1              on failure
```

Example: Get a buffer of 16k for special I/O.

```
#include <stdio.h>
main()
{
char *buf,*sbrk();
     if((buf = sbrk(16*1024)) == (char *)-1) {
          puts("Out of memory"); exit();
     }
     puts("It worked");
}
```

Example: Get all available memory for a buffer.

```
#include <stdio.h>
main()
{
char *buf,*sbrk();
unsigned memsize();
     if((buf = sbrk(memsize())) == (char *)-1) {
          puts("Out of memory"); exit();
     }
     puts("It worked");
}
```

Notes:

- o For UNIX compatibility, restrain n to 32767 and call sbrk() more than once to get the desired amount of memory. This function will work with n an unsigned integer, at the expense of non-portable code.

- o sbrk() checks for text, data and stack over-write. It also looks for attempts to write over file buffers and FCB areas.

- o The return of -1 for failure is normal, but some libraries return 0. Beware as you attempt to port code to new machines.

- o The return of sbrk(0) is the current program break. It does not change pointers - treat it as a report with no action.

- o The symbols end, etext, edata have these meanings:

  end The next usable address after program load.

  etext The physical end of the program text before the file buffers and FCB buffers begin.

  edata Same as etext for C/80 because code and data are in the same place.

- o The externals end, etext, edata are defined in the assembly module ETEXT.C as follows:

```
end:   DW  $END##+TOT_SZ##
edata: DW  $END##
etext: DW  $END##
```

- o The double-# marks a symbol as external for M80, e.g, $END## is equivalent to EXTRN $END.

- o The object code uses addresses 100h to $END, that is, the length of the compiled program is (etext-0x100).

- o TOT_SZ is the total size of the file buffer and file control block (FCB) area that immediately follows the object code. See STDIO.H. The base address of the area is end (= $END). The area uses TOT_SZ bytes with last address (etext - 1).

- o Reference: For end, edata, etext, see Banahan & Rutter, The Unix Book, Wiley Press (1984), p 198. For a description of brk() and sbrk(), see Banahan p 205.

The scanf Function

Purpose:

Parses formatted input text from the stream stdin using a control string to guide the conversion and storage of data items.

Function Header:

```
#include <stdio.h>
int scanf(control,&arg1,&arg2,...)
char *control;                    Control string
&arg1,&arg2,...                   Conversion storage locations.
                                  Pointers do not use the & prefix.
```

The control string is a template for the expected form of the input stream. The function of scanf() is to perform a simple match between the input stream and the control string. Contents of the control will be interpreted during processing as follows:

1. WHITE SPACE. Any white space in the control string causes an indeterminant amount of white space to be skipped in the input. The actual characters skipped will be: SPACE, TAB, CR, LF.

2. NON-CONVERSIONS. Characters other than white space must be matched by the input stream, except for %. The latter will match % if entered in the control string as %%. Otherwise, the % sign begins a block of conversion specifications, and the matching requirement is suspended until the conversion is processed.

3. CONVERSIONS. The allowed conversions parallel printf() usage. Each must start with %, optionally followed by decimal digits which control the maximum width of the conversion, then one, two or three conversion characters:

c	Character, white space too.
s	String, white space delimited, null appended.
[	String, [...] defines acceptable characters.
d	Decimal base 10, signed.
u	Decimal base 10, unsigned.
o	Octal base 8, unsigned.
x	Hexadecimal base 16, unsigned.
b	Binary base 2, unsigned.
e,E	Floating point, signed.
f	Same as e.
g,G	Same as e.
%	Match percent sign.
*	Convert item but do not store result.
ld	Long decimal base 10, signed.
lu	Long decimal base 10, unsigned.
lo	Long octal base 8, unsigned.
lx	Long hexadecimal base 16, unsigned.
lb	Long binary base 2, unsigned.

For example, "%12*ld" means to process at most 12 decimal digits for storage in a long integer location, but skip the storage. The effect is to read over incoming data.

In contrast, "%12ld" does the exact same conversion, but stores the answer at one of the pointers given in the argument list.

The conversion "%[A-Z,a-z]" reads in a string of characters until the character class [A-Z,a-z] is violated. The string is null-delimited after processing stops. Similarly, "%[^A-Z]" negates the character class [A-Z], and causes the read to continue until a character belongs to the class [A-Z]. The most common usage is "[^\r\n]", which is equivalent to the function gets() or LINE INPUT in BASIC.

All floating-point conversions are in single precision, since this library does not have double precision variables.

Long integer conversions are the same as short integer conversions, save the actual size of the final storage location. Short integers are 16 bits and long integers are 32 bits. It is up to the programmer to insure that the control string specifications match the storage size requirements of the argument list.

Returns:

(scanf(...)) is the return value, NOT scanf(). The extra parentheses are required to capture the scanf() return value.

The returned value is EOF if no items were read and EOF was encountered on input.

The returned value is the number of successfully processed items in all other cases. This count should match the number of arguments following the control string.

Example: Read in floating-point numbers from the console.

```
#define pfFLOAT     1
#define sfFLOAT     1
#include <stdio.h>
main()
{ char s[129]; float f;
     do {
          printf("Enter a float: ");
          i = (scanf("%f",&f));
          if(i)
            printf("%10.8f\n",f);
          else {
            printf("Bad input.\nPress RETURN: ");
            gets(s);
          }
     } while(i);
}
```

Example: Read in a string of characters and decode as hexadecimal.

```
#include <stdio.h>
main()
{
char s[129];
int x,i;
   do {
        puts("Enter hex digits");
        i = (scanf("%[^\r\n]",s));
        if((sscanf(s,"%x",&x)) > 0)
          printf("Hex value = %010x, Dec value = %d\n",x,x);
        else
          puts("Bad hex digits");
   } while(i);
}
```

Example: Read in three long integers per line, with comma delimiters.

```
#define pfLONG      1
#define sfLONG      1
#include <stdio.h>
main()
{
char s[129];
int i,j;
long x,y,z;

  do {
       puts("Enter 3 integers separated by commas on one line");
       i = (scanf("%[^\r\n]",s));
       j = (sscanf(s,"%ld, %ld, %ld",&x,&y,&z));
       if(j == 3)
         printf("x = %ld, y = %ld, z = %ld\n",x,y,z);
       else
         puts("Bad input - need 3 integers and 2 commas");
  } while(i);
}
```

Notes:

- o Scanf() calls ungetc(x,fp) to put back the last character, when necessary. You can depend on scanf() to eat white space, in particular newlines.

- o To use long and float features of scanf, write your code as follows:

```
#define sfFLOAT 1        /* include scanf float */
#define sfLONG  1        /* include scanf long  */
#include <stdio.h>
```

  Since the header file processes the sfFLOAT and sfLONG defines, the order is important. The default is to leave off the long and float code for an object code size savings of several kilobytes.

- o The extra parentheses are required under C/80 because of the multiple argument kludge. Failure to include the parentheses will result in a stack address return rather than the number of items processed.

- o It is often safer to use gets() or fgets() to input a line of text and then apply sscanf() to decode the string. The use of ungetc() and full stream I/O sometimes gets in your way, presenting surprises for bad input or end of file conditions.

- o Scanf() often uses basic subroutines that gobble up character classes from the input. Generally, you have to write your own t get it right. Here's an example of how to advance past decimal digits in a string buffer:

```
char *digitpurge(s)
char *s;
{
int x;
     while((x = *s) != EOS) {
          if(isspace(x) || isdigit(x)) ++s;
          else break;
     }
     return s;
}
```

The seek Function

==

Purpose:

Positions the file pointer for an open stream.

Function Header:

```
int seek(fp,offset,type)
FILE *fp;                       Open stream pointer.
int offset;                     Offset from target position,
                                positive, negative or zero.
int type;                       Flag 0,1,2,3,4,5 determines
                                the target position and units:
```

type	Action taken for value 'offset'	Units
0	point to beginning + offset	bytes
1	point to current + offset	bytes
2	point to end + offset	bytes
3	point to beginning + offset	sectors
4	point to current + offset	sectors
5	point to end + offset	sectors

Returns:

0	Seek worked, file pointer reset as per request.
-1	Seek failed.

Example: Seek to the end of an existing file and append a message.

```
#include <stdio.h>
main()
{
FILE *fp,*fopen();
  if(argc <= 1) exit();
  fp = fopen(argv[1],"u");
  if(fp == (FILE *)0) {
    puts("Bad open"); exit();
  }
  if(seek(fp,0,2) == -1) {
    puts("Bad seek"); exit();
  }
  fprintf(fp,"Appended message\n");
  fclose(fp);
 }
```

Notes:

- This set of primitives is used to build the Unix-style fseek() in CLIBU.REL. Use that function to generate code that ports easily to Bell Labs Unix V7.

- A seek of type 2 requires a file size computation, which under CP/M 2.203 is invalid unless the file control block is on disk. To insure the validity of the FCB, seek() writes the FCB back on the disk for a type 2 seek on a file opened for update or write.

- Binary files use sector fill of NULL (00 hex) while others use sector fill of CTRL-Z (1A hex). Carriage return and linefeed translation occur in files not opened in binary mode.

- Seeks on Ascii text files are not dependable (and the code is not portable). For example, moving forward one character may actually move two, because the first was a carriage return (0D hex).

- A rewind or seek to end of file is dependable and portable for ASCII or BINARY files. Likewise, a move by whole sector is Ok, but offsets into a sector can be off by the number of CR/LF pairs in that sector.

- **DO NOT USE the SEEK on the C/80 distribution disk.** It won't link (use that SEEK with Walt Bilofsky's CLIBRARY.REL).

The seekend Function

==

Purpose:

Seeks to the end of an open file given by descriptor fd. The C/80 file buffer pointer is positioned to the next byte to be written. This is a special C/80 function needed to make the Unix library perform as expected. Not intended to be portable.

Function Header:

```
seekend(fd)
int  fd;                    File descriptor
```

Returns:

Nothing. It either works or CP/M will bounce you to the system.

Example: Seek to the end of a file in a ROM application.

```
#include <stdio.h>
myopen(file)
char *file;
{
int fd;

     fd = fopen (file,"r");          /* use C/80 fopen() */
     if(fd != -1) seekend(fd);       /* and unix seekend */
     return fd;
}
```

Notes:

o If the file was opened in binary mode, then the seek is to a record boundary.

o If the file was opened in Ascii text mode, then the buffer is filled from the last disk record and the file is positioned to the ctrl-Z at the end of the file or the next new record.

o This version is used in the Unix fopen() source to reduce the seek() overhead. It saves about 700 bytes in code size.

o The following externs from IOTABLE.CC are used:

```
extern  int
IOind[],IOsect[],IOfcb[],IOend[],IOsize[],IObuf[];
extern  char
IObin[];
```

o File descriptors are integer offsets into the above arrays. Here, stream pointers and file descriptors are identical.

The seldsk Function

==

Purpose:

Selects a disk drive under CP/M.

Function Header:

```
VOID seldsk(x)
int x;                  Drive number 0...15 for drives A: to P:
```

Returns:

Nothing useful. The action taken is to mount the disk, read the file allocation bitmap to main memory and mark the disk as $R/W. All other drives are left untouched.

Example: Let the user change disks in drive C:.

```
#include <stdio.h>
main()
{
    log("C:");
}

log(drive)
char *drive;
{
char s[129];
  printf("Insert a new disk in %s and hit RETURN: ",drive);
  gets(s);
  reset();
  seldsk(toupper(drive[0])-'A');
}
```

Notes:

- o Seldsk() is not a standard function. It is handy for writing interactive CP/M software.

- o To simulate seldsk() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>                     #include <stdio.h>
seldsk(x) int x;                       seldsk(x) int x;
{                                      {
  return bdos(14,x);                     return ccBDOS(x,14);
}                                      }
```

The setatt Function

Purpose:

Set the CP/M file attributes, which are:

Read-only	$R/O or $R/W
System	$SYS or $DIR
Archive	Used by some backup programs, but not by CP/M at present.

Function Header:

```
int  setatt(name,code)
char *name;                 File to update, null-terminated string
int  code;                  Coded attribute word, 0 <= n <= 7:

                            code&1 = Read-only attribute
                            code&2 = System attribute
                            code&4 = Archive attribute
```

Returns:

0	Attributes set as requested
-1	File not found or set failed due to write-protect tab or disk directory write error.

Example: Set the attributes of a file entered on the command line.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int n;
    if(argc <= 2) { puts("Usage: A>main filename code"); exit(); }
    n=atoi(argv[2]);
    if(n < 0 || n > 7) {
        puts("Bad code, range is 0 to 7"); exit();
    }
    if(setatt(argv[1],n)) {
        puts("File not found or disk write-protected"); exit();
    }
    printf("File %s set to %s,%s,%s\n", argv[1],
           (n&1 ? "$R/O":"$R/W"), (n&2 ? "$SYS":"$DIR"),
           (n&4 ? "Archive":"Non-Archive") );
}
```

Notes:

o The name is a normal C string with NULL delimiter.

o The coded word uses only bits 0,1,2.

The setbuf Function

Purpose:

Changes the location of the current file buffer to the given address.

Function Header:

```
int  setbuf(fp,buffer)
FILE *fp;                  Stream pointer, open stream
char *buffer;              Base address of the data area
                           to be used for a file buffer
```

Example: Change buffer size to 4096 and change buffer to a new location for further I/O.

```
#include <stdio.h>
setbsize(n,fd)
int n,fd;
{
 extern int IOsize[];
 extern MAXCHN[];
        if(1 <= fd && fd < MAXCHN) IOsize[fd] = n;
        else exit();
}
main()
{
extern int IOsize[];
int x;
FILE *fp,*fopen();
char buf[4096];

    if(fp = fopen("TEST","r")) {
        setbsize(4096,fp);
        setbuf(buf,fp);
        while((x=getc(fp)) != EOF) putchar(x);
    }
}
```

Notes:

- o Stream files always have buffers set up by a call to sbrk(). You can run out of memory by using large file buffers that are not de-allocatted after use.
- o The buffer size is fixed to the value in IOsize[fp]. You may change it in C/80, but note that this creates non-Unix source code.
- o Change the file buffer location to where the current data buffer is located.

The setjmp and longjmp Functions

Purpose:

Provides a non-local GOTO which allows a graceful and certain exit from deeply nested procedures. Used primarily for backing out out a sequence of function calls during which a fatal error was encountered.

Function Header:

```
#include <setjmp.h>
int setjmp(env)
jmp_buf env[1];                Environment save area. Definitions are
                               in SETJMP.H header file.

VOID longjmp(env,v)
jmp_buf env[1];                Environment save area. Definitions are
                               in SETJMP.H header file.
int v;                         Value for longjmp() to return.
```

Returns:

setjmp:	0	Normal return.
longjmp:	v	Restores registers from env[], loads CPU register with value v. Processor jumps to last instruction in SETJMP (a RET).

Explanation of the example: The program below is an infinite loop that accepts keyboard characters.

Pressing certain keys causes an invisible GOTO to parts of main(). In this sense, setjmp() makes an invisible program LABEL and longjmp() is a GOTO LABEL.

The difference between this method and using actual labels is that the GOTO may act from within deeply nested functions with target label in another module.

Example: Test program to observe effect of setjmp(), longjmp().

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf env1[1];
jmp_buf env2[1];

f(fp)
FILE *fp;
{ int x;
    putchar(':');
    x = getcbinary(fp); putchar('\n');
    switch(x) {
    case 'Z'-'@': longjmp(env1,10);
    case '\n'   : longjmp(env1,20);
    case 'C'-'@': longjmp(env1,30);
    case 'A'-'@': longjmp(env2,1);
    }
}

main() {
  puts("Setup for env1");
  switch(setjmp(env1)) {
    case 0: break;
    case 10: puts("EOF encountered on input");
    case 20: puts("Linefeed in input"); break;
    case 30: puts("Ctrl-C normal exit"); exit();
    default: puts("Undefined error");
  }
  puts("Setup for env2");
  switch(setjmp(env2)) {
    case 0: break;
    case 1: puts("ctrl-A struck"); break;
    default: puts("Undefined error");
  }
  puts("Special characters are ctrl-A, ctrl-C, ctrl-J, ctrl-Z");
  while(TRUE) f(stdin);
}
```

Notes:

- Note that **jmp_buf** varies among systems as does the storage class of the array. All this info is hidden in the SETJMP.H header file, whose exact contents varies between systems.

- The definition of **jmp_buf** below is made in SETJMP.H in order to present a portable interface across compilers. The size of the structure is set at 4 to protect the STACK POINTER and RETURN ADDRESS that were in force when setjmp() was called to load the environment array. Note that PSW, BC, DE, HL are all clobbered by setjmp(). Longjmp() loads register HL with value v and sets the stack pointer to its old state.

```
        struct jmpbuffer { char *STACKP; char *RETADR;};
        #define jmp_buf struct jmpbuffer
```

- Here is a common error:

```
   WRONG WAY                  RIGHT WAY
   jmp_buf env;               jmp_buf env[1];
   setjmp(env);               setjmp(env);
```

  The compiler will complain about the error, so there is not much chance of it sneaking by the programmer.

- Portability of the C/80 code is not changed by adding the [1]. On a system using typedef, the result is

```
              struct jmpbuffer env[1][1];
```

  instead of

```
              struct jmpbuffer env[1];
```

  These statements generate exactly the same addressing mode and storage requirements.

The sfree Function

==

Purpose:

Frees memory that has been obtained by calls to core() or sbrk().

Function Header:

```
unsigned sfree(n)
unsigned n;                  Amount of memory to free. The maximum
                             value of n is 65535. See notes below.
```

Returns:

The amount freed, which is n if it worked as planned, 0 if it failed completely, or some unsigned integer in between, which represents the function's attempt to please the caller within the restraints placed by the location of file buffers and file control blocks.

Example: Free up all the space assigned by sbrk() and core() after the the program is finished, so that a re-start will not run out of memory.

```
#include <stdio.h>
main()
{
char *p,*core();
     p = core(32767);
     fillchr(p,'A',32767);
     sfree(-1);
     puts("It worked");
}
```

Notes:

- File buffers and file control block areas will not be freed by sfree().

- The dynamic building of file buffers and file control blocks may leave large areas of memory unusable by sbrk() or sfree(). Avoid this problem by allocating enough built-in file buffers for the program to run without having to call sbrk() for more space.

- This function will remove orphan buffers from the heap if followed by a call to sfree:

```
unbuff(fd) int fd;
{
extern char IOch[]; extern int IOfcb[],IObuf[];
   if(IOch[fd] == -1) IOfcb[fd] = IObuf[fd] = 0;
}
```

shells

The shellsort Function

Purpose:

Shellsort for an array of character strings with easy-to-change comparison method. Changes pointers but not actual data in the string.

Function Header:

```
shells(p,n,CMP)
char *p[];               Base address of pointer array
int  n;                  Number of array elements to sort
int  (*CMP)();           Compare routine for two strings p,q
                         that returns an integer with the
                         same rules used by Unix strcmp(p,q):

                            = 0    strings p,q are equal
                            < 0    p < q
                            > 0    p > q
```

Returns:

Nothing of value

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
        shells(p,10,strcmp);
        for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a             Numerals are less than letters
1. 1, C             in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3             UPPERCASE is less than lowercase
8. a, 8             in the ASCII standard
9. d, 5
```

Notes:

- o You can most often use strcmp() as the argument for cmp().

- o It is necessary to declare the function cmp() before the call to shells(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.

- o To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example above.

- o This source file is very portable. Use it on any system with minimal C compiler.

Reference:

The C Programming Language, p 116,
by Kernighan & Ritchie
Prentice-Hall, Englewood Cliffs (1978)

The signal Function

==

Purpose:

Connects a C function with CPU interrupt processing. Currently a CP/M error return that allows code using Signal() to compile with no errors.

Function Header:

```
int (*signal (sig,f))()
int  sig;               Exception number to trigger signal().
int  (*f)();            C function to call when exception
                        number sig is encountered by the
                        interrupt system.
```

Returns:

0	No error
-1	Argument sig not in range (present default).

Library routines preserve registers and condition codes. Function f() exits to the interrupted C process and normal program execution resumes.

Notes:

- o Double translation of the number sig is required in order for signal() to set up an interrupt.
- o At present, no implementation of signal() has been made. It is an error return to allow Unix source code to compile without errors, even though it will probably not run.
- o Signal is a stub function like the other Unix System III compatibility functions. See UNIXIO for a complete list and Banahan & Rutter, The Unix Book, Wiley Press(1983) for function.

The sin Function

Purpose:

Compute the sine of real float value x. The value x is assumed in radians (not degrees).

Function Header:

float sin(x)	Sine.
float x;	Float value for computation.

Returns:

number	Answer between -1 and 1 No error checking for large x

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sin();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sin(%f) = %f, errno = %d\n",x,sin(x),errno);
}
```

Notes:

o Use deg() and rad() for conversions.

o Method used is Rational approximation from Cheney & Hart. Coefficients from #3370, C&H.

o Cody & Waite p 137 had too many errors with the current float package. Went back to Cheney & Hart to get it right.

o The switch case below uses more code but runs about 32% faster than other methods. Use ASM to optimize.

o No error checks for large x. The max size of x is about 1440 degrees, which is 4 wraps of the unit circle (25.13 radians).

o No assembler optimization.

The sinh Function

Purpose:

Compute the hyperbolic sine of real float value x.

Function Header:

float sinh(x)	Hyperbolic sine $(e^{x} - e^{-x})/2$.
float x;	Positive, negative or zero argument.

Returns:

number	x in range
INF	x too large positive (see notes).
-INF	x too large negative (see notes).
	errno = ERANGE returned to flag error.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sinh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sinh(%f) = %f, errno = %d\n",x,sinh(x),errno);
}
```

Notes:

o Method used is sinh(x) = (exp(x) - exp(-x))/2.0.

o The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The sob Function

==

Purpose:

Skips over leading white space. Stands for Skip Over Blanks.

Function Header:

```
char *sob(s)
char *s;                     String, null-terminated.
```

Returns:

The address of the null terminator in the string, or the address of the first non-blank character. Blanks are defined by isspace().

Example: Find and print the first word in a line of text.

```
#include <stdio.h>
main()
{
char buf[129];
char *s,*sob();
     printf("Enter a line of text: ");
     gets(buf);
     s = sob(buf);
     while(*s) {
          if(isspace(*s)) break;
          putchar(*s++);
     }
     putchar('\n');
}
```

Notes:

o This function is almost always used in conjunction with fnb(), in order to isolate tokens in a line of text.

o High-overhead functions like scanf() can often be avoided in a portable way by using sob() and fnb().

o Both sob() and fnb() return the address of the null delimiter of the string in case the character class checking runs off the end of the string.

o The source code for sob() is:

```
char *sob(s) char *s;
{
  while(isspace(*s)) ++s; return s;
}
```

The sprintf Function

==

Purpose:

Outputs formatted data to a string in memory using a control string and an appropriate argument list of variable length.

Function Header:

```
     sprintf(s,control,arg1,arg2,...);
     char *s;                            String in memory.
     char *control;                      control string, see below
     arg1,arg2,...                       appropriate arguments,
                                         8, 16 or 32-bit data, see below
```

The control string requirements are the same as for printf. See the printf() rules.

Returns:

Nothing.

Example: Print a number in Decimal, Octal, Hex, Binary to a string.

```
     #include <stdio.h>
     main()
     {
     char s[200];
       int i;
       for(i=0;i<128;++i)
         sprintf(s,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
     }
```

Example: Use the denser sprintf in the preceding example.

```
     #define libl CLIBM
     #include <stdio.h>
     #undef sprintf
     #define sprintf prnt_1(),prnt_3
     main()
     {
     char s[200]; int i;
       for(i=0;i<128;++i)
         sprintf(s,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
     }
```

Notes:

o In this library, sprintf() is supplied in two versions. The fast version lacks some of the features found in the denser version for longs and floats. Both versions support multiple arguments. Neither version is fully recursive.

The sqrt Function

==

Purpose:

Compute the square root of real float value x

Function Header:

```
float sqrt(x)
float x;             Positive or zero argument for the square root.
```

Returns:

```
number          x >= 0 returns the square root of x.
0.0             x < 0 error
                errno = EDOM returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sqrt();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sqrt(%f) = %f, errno = %d\n",x,sqrt(x),errno);
}
```

Notes:

o Newton's method as here, or use sqrt(x) = pow(x,.5);

o Method from C&H, 6.1.3, 6.1.7. Startup from table 0231, C&H.

The sscanf Function

==

Purpose:

Parses formatted input text from a string in memory and stores the answers into given memory locations.

Function Header:

```
#include <stdio.h>
int sscanf(s,control,&arg1,&arg2,...)
char *s;                                String to be parsed.
char *control;                          Control string, see below
&arg1,&arg2,...                         Appropriate arguments
                                        See below for rules.
```

Returns:

The number of successfully parsed arguments. An error occurred if the number returned does not match the number of arguments following the control string. C/80 requires that sscanf() be enclosed in parentheses in order to check the return value, i.e.,

```
               i = (sscanf(s,"%d",&x));      RIGHT WAY
rather than
               i = sscanf(fp,"%d",&x);       WRONG WAY
```

Example: The following reads numbers from the console until either a data file error occurs or end of file is reached. Only one number per line is allowed.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
char *gets();
char s[129];
int x;
     while(gets(s) != (char *)0) {
        if((sscanf(s,"%d",&x))>0) printf("%d\n",x);
        else break;
     }
}
```

Notes:

- o The scanf family of functions accepts addresses only for its argument list. To error-check coding, verify that each argument has the address operator & as a prefix, or else the argument is a pointer (hence already an address).

- o The converted values are stored at the given addresses in order left to right. Skips in the control string do not have an argument so the argument count may not match the conversion count.

- o Short counts may hang the run-time package. Overly abundant counts will leave variables unfilled at run time.

- o Always check the return of sscanf() to see if matches the expected value. It is easy to program infinite loops using sscanf().

- o C/80 and its multiple-argument kludge cause users to write parentheses around sscanf() in order to recover the returned value. For example

```
if((sscanf(s,"%d",&x)) > 0)
```

  will not work under C/80 with the extra parentheses removed.

- o To turn on float or long libraries for use with sscanf(), use the appropriate switches:

```
#define sfLONG  1         /* turn on long sscanf */
#define sfFLOAT 1         /* turn on float sscanf */
```

  The compiled code will change in size according to how much of the float and long libraries are actually used.

The ssort2 Function

==

Purpose:

Shell-Metzner sort in assembler for strings of fixed length which reside in an array of string pointers.

Function Header:

```
VOID ssort2(size,table,reclen)
int size;                      Number of pointers to sort.
char *table[];                 Table of string pointers.
unsigned reclen;               Length of each string;
```

Returns:

Nothing. Pointers are altered to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of names of fixed length.

```
#include <stdio.h>
main()
{
int i;
static
char *names[] = {
     "DAN RATHER  ",
     "E.G.MARSHALL",
     "JIMMY CARTER",
     "DONALD KNUTH"
};

  ssort2(4,names,12);
  for(i=0;i<4;++i) puts(names[i]);
}
```

Notes:

- o The order of the arguments is essential.
- o A fixed record length is required. For variable-length records see shell() and quick() in CLIBU.REL or ssort3() below.

The ssort3 Function

==

Purpose:

Shell-Metzner sort in assembler for strings of variable length which reside in memory. Access is by a special structure that simulates the string storage in Microsoft MBASIC.

Function Header:

```
struct mstr {
     char length; char *string;
};

VOID ssort3(size,&table)
int size;                      Number of pointers to sort.
struct mstr table[];           Table of string pointers.
```

Returns:

Nothing. Pointers are altered to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of names of variable length stored as MBASIC strings.

```
#include <stdio.h>
main()
{
int i;
struct mstr {
     char length; char *string;
};
static
struct mstr names[] = {
     {10,"DAN RATHER"  },
     {12,"E.G.MARSHALL"},
     {12,"JIMMY CARTER"},
     {12,"DONALD KNUTH"},
     { 9,"TOM JONES"   },
     { 8,"007 BOND"    }
};
  ssort3(6,&names);
  for(i=0;i<4;++i) puts(names[i].string);
}
```

Notes:

- o The order of the arguments is essential.
- o The algorithm is Shell-Metzner's sort, with the strings left in place in memory. The table is re-organized to sorted order on return.

The stdin, stdout, stderr Streams

Purpose:

Defines the file pointer for standard input, standard output and standard error messages. Full re-direction capabilities are supported. No pipes or filters.

Function Header:

The definitions appear in STDIO.H, by C convention.

```
#define stdin (FILE *)fin__()
#define stdout (FILE *)fout__()
#define stderr (FILE *)252
```

Returns:

The stream pointer for the indicated stream, allowing for re-direction. No re-direction is possible for the standard error stream. It is always the console.

Example: Check the standard input stream to see if indirection is in force.

```
#include <stdio.h>
main()
{
     if(stdin == (FILE *)0)
          fprintf(stderr,"Stdin = console\n");
     else
          fprintf(stderr,"Stdin is file descriptor %d\n",stdin);
}
```

Notes:

- o This implementation of stdin, stdout and stderr is at the macro preprocessor level, therefore it generates no object code in the COM file. Linkage, in case of actual use, causes module FINFOUT.REL to be linked, which adds 8 bytes to the object code. The routines fin__() and fout__() simply return the stream descriptors fin and fout.

- o The stream stderr is hard-wired to the console. You can change it by re-assembly of the C program, but not by command line options.

- o The descriptors fin and fout are set to zero at the start of a program. Re-direction alters these descriptors. Note that fin or fout can be closed by the user. After doing so, the user should set fin = fout = 0.

The strcat Function

Purpose:

Appends a null-terminated string str2 to the null-terminated string str1.

Function Header:

```
char *strcat(str1,str2)
char *str1,*str2;              Source strings, null-terminated.
```

Returns:

```
str1           The base address of the destination string.
```

Example: Fix a CP/M file name.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[256];
char *index();
     if(argc <= 1) exit();
     strcpy(s,argv[1]);
     if(index(s,'.') == (char *)0) strcat(s,".COM");
     printf("s = %s\n",s);
}
```

Notes:

o Strcat() is dangerous. The classic error is to append a string to a character pointer that is not null-terminated.

o Storage over-run is possible. There is no run-time check for writing beyond the limits of storage.

o See also strcpy, strncpy(), strncat() and moveMEM().

o Here is the standard strcat() code:

```
char *strcat(s1,s2) char *s1,*s2;
{
char *p;
  p = s1;
  while(*s1) ++s1; while(*s1++ = *s2++) ; return p;
}
```

The strchr Function

==

Purpose:

Searches for character c inside string s.

Function Header:

```
char *strchr(s,c)
char *s,c;                  Source string s and character c to find.
```

Returns:

```
(char *)0                   no match
&s[i]                       address of first match to s[i] == c
&s[n]                       n=strlen(s), if c = '\0'
```

Example: Find the end of a C string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static
char *s = "This is a Test";
char *p;
char *strchr();
     p = strchr(s,'\0');
     printf("Search for null at end of a string s.\n");
     printf("s = %u, p = %u, strlen(s) = %u\n",s,p,strlen(s));
}
```

Notes:

o strchr() is the same as index().

o Here is the standard strchr code:

```
  char *strchr(s,c) char *s,c;
  {
     do {
        if(*s == c) return s;
     } while(*s++);
     return (char *)0;
  }
```

strcmp

The strcmp Function

Purpose:

Compares two null-terminated strings for equality, less than or greater than.

Function Header:

```
int strcmp(str1,str2)
char *str1,*str2;              Source strings, null-terminated.
```

Returns:

< 0	for str1 < str2
= 0	for str1 = str2 (same length and characters)
> 0	for str1 > str2

Example: Test a string for a match to one on the four CP/M devices.

```
#include <stdio.h>
main()
{
int i,j; char s[129];
static char *device[] = { "CON:","LST:","RDR:","PUN:" };
     printf("Enter a device name: ");
     gets(s); cvupper(s);
     for(i=0;i<4;++i) {
          if(strcmp(device[i],s)==0) {
               printf("Device match is: %s\n",device[i]);
               printf("Mismatches:\n");
               for(j=0;j<4;++j) {
                    if(j != i) puts(device[j]);
               }
               return;
          }
     }
     puts("No match");
}
```

Notes:

- o Strcmp() does what is expected when one or both of the strings are null strings. Other cases of interest: "abc" > "ABC", "abc" < "abcd".

- o Standard strcmp() source code:

```
int strcmp(s1,s2) char *s1,*s2;
{
  while(*s1 == *s2) { if(*s1 == '\0') return 0; ++s1; ++s2; }
  return (*s1 - *s2);
}
```

The strcpy Function

Purpose:

Copies a null-terminated string str2 to the storage area given by the string str1. The destination string is also null-terminated.

Function Header:

```
char *strcpy(str1,str2)
char *str1;                     Destination string.
char *str2;                     Source string, null-terminated.
```

Returns:

```
str1          The base address of the destination string.
```

Example: Copy a command line argument to safety.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[256];
     if(argc <= 1) exit(); strcpy(s,argv[1]);
     printf("argv[1] = %s\n",s);
}
```

Example: Copy lines of a file into an array.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i,n;
FILE *fp,*fopen();
char buf[129],*s[256];
char *core(),*fgets();
     if(argc <= 1) {
          puts("Usage: A>load filename"); exit();
     }
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Bad open"); exit();
     }
     i = 0;
     while(i<256 && fgets(buf,128,fp) != (char *)0) {
          s[i] = core(1+strlen(buf));
          strcpy(s[i],buf);
          ++i;
     }
     fclose(fp);
     n = i;
     puts("The saved lines:");
     for(i=0;i<n;++i) printf("s[%d]: %s\n",i,s[i]);
}
```

Notes:

- o Strcpy() is dangerous because there is no check for buffer limits.

- o Besides storage over-run caused by insufficient estimate of the requirements, there is the problem on uninitialized data:

```
          char *s;
          strcpy(s,t);
```

  The problem here is that s is initialized to 0, the base of the operating system. The strcpy() code above writes string t to address 0. This may cause a CP/M system to crash. To fix the problem, use core() to get storage for s, or use an auto array:

```
  char *s;                       char *s;
  char buf[256];                 char *core();
  s = buf;                       s = core(256);
  strcpy(s,t);                   strcpy(s,t);
```

- o See also strncpy() and moveMEM().

- o Standard strcpy source code:

```
char *strcpy(s1,s2) char *s1,*s2;
{
char *p;
  p = s1; while(*s1++ = *s2++) ; return p;
}
```

The strcspn Function

==

Purpose:

Searches a null-terminated string s for the first match to any character in the null-terminated string called set.

Function Header:

```
int strcspn(s,set)
char *s;                  Target string s to be searched.
char *set;                Possible characters to match.
```

Returns:

```
strlen(s)       No match occurred.
i               The first index i where s[i] matches a character
                of string set.
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return strlen(s), otherwise return the first index n for which s[n] is in B.

Example: Find the first digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789"; int i;
     if(argc > 1) {
       if((i = strcspn(argv[1],set)) < strlen(argv[1]))
         printf("Matched at offset %d\n",i);
     }
}
```

Notes:

o Standard strcspn source code:

```
int strcspn(s,set) char *s,*set;
{
int n; char *p;
  n = 0;
  while(s[n]) {
    p = set;
    while(*p) {
      if(*p++ == s[n]) return n;
    }
    ++n;
  }
  return n;
}
```

The strlen Function

Purpose:

Computes the integer length of a null-terminated string.

Function Header:

```
int strlen(str)
char *str;          Source string, null-terminated.
```

Returns:

```
0              String has no characters (empty string "").
n              String has n characters.
```

Example: Find the length of a user-entered token.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
     if(argc <= 1) exit();
     printf("argv[1] = %s, strlen(argv[1]) = %d\n",
             argv[1], strlen(argv[1]) );
}
```

Notes:

- o strlen() returns an unsigned integer. If you get a negative value for strlen(), then suspect a program bug. It is quite unusual to have a C string in excess of 32767 bytes.

- o Strlen() is often used to find the null position in a C string. If p is a string pointer, then p += strlen(p) computes a pointer to the null terminator of the string.

- o Standard strlen source code:

```
int strlen(s)
char *s;
{
int n;
  n = 0;
  while(s[n]) ++n;
  return n;
}
```

The strncat Function

Purpose:

Appends a null-terminated string str2 to the storage area given by the string strl. At most n bytes are transferred. The destination string is ALWAYS null-terminated.

Function Header:

```
char *strncat(strl,str2,n)
char *strl,*str2;             Source strings, null-terminated.
int n;                        Number of bytes to transfer.
```

Returns:

```
strl          The base address of the destination string.
```

Example: Fix a CP/M file name.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[25]; char *index();
     if(argc <= 1) exit();
     if(p = index(s,':')) s[0] = '\0';
     else strcpy(s,"A:");
     strncat(s,argv[1];20); printf("s = %s\n",s);
}
```

Notes:

- o Strncat() is dangerous. The classic error is to append a string to a character pointer that is not null-terminated.

- o The following problem can occur with strncat:

```
          char *s;
          strncat(s,t,1+strlen(t));
```

  The problem here is that s is initialized to 0, so strncat() will look for the end of the string that starts at the operating system base 0. Surely it will be found, then strncat appends str2. The end result is a very unpredictable system crash.

- o Standard strncat() source code:

```
char *strncat(s1,s2,n) char *s1,*s2; int n;
{ char *p;
  p = s1; while(*s1) ++s1;
  while((n > 0) && ((*s1++ = *s2++) != '\0')) --n;
  *s1 = '\0'; return p;
}
```

The strncmp Function

===

Purpose:

Compares two null-terminated strings for equality, less than or greater than. During comparison, at most n characters are considered.

Function Header:

```
int strncmp(str1,str2,n)
char *str1,*str2;              Source strings, null-terminated.
int n;                         Maximum comparison count.
```

Returns:

< 0	for str1 < str2
= 0	for str1 = str2 (same length and characters)
> 0	for str1 > str2

Example: Test a string for a match to one on the four CP/M devices.

```
#include <stdio.h>
main()
{
int i,j; char s[129];
static
char *device[] = { "CON:","LST:","RDR:","PUN:" };
     printf("Enter a device name: ");
     gets(s); cvupper(s);
     for(i=0;i<4;++i) {
          if(strncmp(s,device[i],4)==0) {
               printf("Device match is: %s\n",device[i]);
               printf("Mismatches:\n");
               for(j=0;j<4;++j) {
                    if(j != i) puts(device[j]);
               }
               return;
          }
     }
     puts("No match");
}
```

Notes:

o Strncmp() is strcmp() with a restraint. Standard source code:

```
int strncmp(s1,s2,n) char *s1,*s2; int n;
{
  while((n-- > 0) && (*s1 == *s2)) {
    if(*s1 == '\0' || n == 0) return 0; ++s1; ++s2;
  }
  return (*s1 - *s2);
}
```

The strncpy Function

==

Purpose:

Copies a null-terminated string str2 to the storage area given by the string strl. Exactly n bytes will be transferred, with null fill used if str2 is less than n bytes in length.

Function Header:

```
     char *strncpy(strl,str2,n)
     char *strl,*str2;              Source strings, null-terminated.
     int n;                         Number of bytes to transfer.
```

Returns:

```
     strl          The base address of the destination string.
```

Example: Copy a command line argument to safety.

```
     #include <stdio.h>
     main(argc,argv)  int argc; char **argv;
     {
     char s[11];
          if(argc > 1 && strlen(argv[1]) > 10) {
               printf("Bad token length"); exit();
          }
          strncpy(s,argv[1],1+strlen(argv[1]));
          printf("argv[1] = %s\n",s);
     }
```

Notes:

- o Strncpy() is dangerous. An error can be made in its usage which allows the destination string to be non-terminated (no null at the end). Strncpy() will not write a null terminator unless strlen(str2)<n.

- o The following code will over-write the operating system because the initial value in s is 0:

```
                    char *s;
                    strncpy(s,t,1+strlen(t));
```

- o Conversion of random file routines from Microsoft's MBASIC can use strncpy() to write a field in a random record with 0 fill.

- o Standard strncpy() source code:

```
  char *strncpy(s1,s2,n) char *s1,*s2; int n;
  { char *p; p = s1;
    while(n-- > 0) { *s1++ = ( (*s2) ? *s2++ : '\0'); } return p;
  }
```

strpbrk

The strpbrk Function

Purpose:

Searches a null-terminated string s for the first match to any character in the null-terminated string called set.

Function Header:

```
char *strpbrk(s,set)
char *s;                   Target string s to be searched.
char *set;                 Possible characters to match.
```

Returns:

(char *)0	No match occurred.
&s[i]	Address of the the first match.

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return NULL, otherwise return the first pointer &s[n] for which s[n] is in B.

Example: Find the first digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789";
char *p,*strpbrk();
     if(argc > 1) {
       if(p = strpbrk(argv[1],set))
         printf("argv[1] = %u, match at %u\n",argv[1],p);
     }
}
```

Notes:

o Standard strpbrk source code:

```
char *strpbrk(s,set)
char *s,*set;
{
char *p;
  while(*s) {
    p = set;
    while(*p) {
      if(*p++ == *s) return s;
    }
    ++s;
  }
  return (char *)0;
}
```

The strpos Function

===

Purpose:

Scan null-terminated string s for a match to character c.

Function Header:

```
int strpos(s,c)
char *s;                    Target string s to be searched.
char c;                     Character c to locate.
```

Returns:

-1	failure
strlen(s)	if c == '\0'
n	n is the first index with s[n] == c

Example: Find the first letter 'A' in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i;
    if(argc > 1) {
      if((i = strpos(argv[1],'A')) != -1)
        printf("argv[1] == 'A' at offset %u\n",i);
    }
}
```

Notes:

o This is the integer offset version of the function index().

o Standard strpos source code:

```
int strpos(s,c)
char *s,c;
{
int n;
  n = 0;
  do {
       if(s[n] == c) return n;
  } while(s[n++] != '\0');
  return -1;
}
```

The strrchr Function

==

Purpose:

Search null-terminated string s for the last match to character c.

Function Header:

```
char *strrchr(s,c)
char *s;                  Target string s to be searched.
char c;                   Character c to locate.
```

Returns:

(char *)0	No character of s matches c.
&s[i]	i is the largest index for which s[i] == c

Example: Find the last occurrence of letter 'A' in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char *p,*strrchr();
     if(argc > 1) {
       if((p = strrchr(argv[1],'A')) != (char *)0)
         printf("Last argv[1] == 'A' at address %u\n",p);
     }
}
```

Notes:

o This is the same function as rindex().

o Standard strrchr source code:

```
char *strrchr(s,c)
char *s,c;
{
int n;
  n = 0;
  while(s[n]) ++n;
  do {
       if(s[n] == c) return (&s[n]);
  } while(n--);
  return (char *)0;
}
```

The strrpbrk Function

==

Purpose:

Search null-terminated string s for the last match to any character in the null-terminated string called set.

Function Header:

```
char *strrpbrk(s,set)
char *s;                 Target string s to be searched.
char *set;               String of possible matching characters.
```

Returns:

```
(char *)0           no match occurred, otherwise
&s[i]               address of the LAST match
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return NULL, otherwise return the last pointer &s[n] for which s[n] is in B.

Example: Find the last occurrence of a digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789";
char *p,*strrpbrk();
  if(argc > 1) {
    if((p = strrpbrk(argv[1],set)) != (char *)0)
      printf("Last digit of argv[1] = %u at %u\n",argv[1],p);
  }
}
```

Notes:

o Standard strrpbrk source code:

```
  char *strrpbrk(s,set)
  char *s,*set;
  {
  char *p;
  int n;
    n = 0;
    while(s[n]) ++n;
    do {
        p = set;
        while(*p) { if(*p++ == s[n]) return (&s[n]); }
    } while(n--);
    return (char *)0;
 }
```

The strrpos Function

Purpose:

Reverse scan null-terminated string s for a match to character c.

Function Header:

```
int strrpos(s,c)
char *s;                 Target string s to be searched.
char c;                  Character c to match.
```

Returns:

-1	Failure.
n	n is the largest index with s[n] == c.
strlen(s)	If c = '\0'.

Example: Find the last occurrence of character 'A' in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i;
  if(argc > 1) {
    if((i = strrpos(argv[1],'A')) != -1)
      printf("Last argv[1] == 'A' at offset %d\n",i);
  }
}
```

Notes:

o Standard strrpos source code:

```
int strrpos(s,c)
char *s,c;
{
int n;
  n = 0;
  while(s[n]) ++n;
  do {
       if(s[n] == c) return n;
  } while(n--);
  return -1;
}
```

The strspn Function

==

Purpose:

Search null-terminated string s for the first character that is not in the string called set.

Function Header:

```
int strspn(s,set)
char *s;                     Target string for testing.
char *set;                   String of characters to skip over.
```

Returns:

```
strlen(s)      All chars in s are also in set.
n              n is the first index with s[n] not in set.
0              String set is a null string.
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If all characters of s are in B, then return strlen(s), otherwise return the smallest index n for which s[n] is not in B. (B is empty if set = "\0")

Example: Find the first non-digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789"; int i;
  if(argc > 1) {
    if((i = strspn(argv[1],set)) != 0 && i < strlen(argv[1]))
      printf("First non-digit of argv[1] is at offset %d\n",i);
  }
}
```

Notes:

o Standard strspn source code:

```
int strspn(s,set) char *s,*set;
{
char *p; int n;
  n = 0;
  if(*set == '\0') return 0;
  while(s[n]) {
    p = set;
    while(*p) {
      if(*p++ == s[n]) break; if(*p == '\0') return n;
    }
    ++n;
  }
  return n;
}
```

The swab Function

==

Purpose:

Copies one area of memory to another, reversing pairs of bytes. The number to copy must be even.

Function Header:

```
int  swab(s,d,n)
char *s;                  Char pointer to source data area
char *d;                  Char pointer to destination data area
int  n;                   Integer number of bytes to copy
```

Returns:

```
0              Always.
```

Example: Sample program to swap halves of words in an array.

```
#include <unix.h>
#include <stdio.h>
char *p = "ababababababababababababababababababababababababab";
char *q = "cccccccccccccccccccccccccccccccccccccccccccccccccc";
main(argc,argv)
int argc;
char *argv[];
{
int i;
        i = strlen(p);
        puts(p);
        puts(q);
        swab(p,q,i);
        puts(q);
}
```

Output from the above main program:

```
A>main
ababababababababababababababababababababababababab
cccccccccccccccccccccccccccccccccccccccccccccccccc
babababababababababababababababababababababababababa
A>
```

Notes:

o Copies data from a reverse byte-sex machine. For example, 68000 CPU integer data is reversed from the Intel Reverse Format found on all Z80, 8080, 8088, 8086, 80186, 80286 processors.

The system Function

==

Purpose:

Pass a command string to the console command processor.

Function Header:

```
int  system(s)
char *s;                CP/M command lines to run, string is
                        null-terminated. See example below.
```

Returns:

```
Never           On success, exits to system
0               On failure
```

Example: To do the CP/M commands:

```
CC TEST
M80 =TEST
L80 TEST,TEST/N/E
ERA TEST.MAC
```

Place this this code in your C program:

```
system("CC TEST\nM80 =TEST\nL80 TEST,TEST/N/E\nERA TEST.MAC");
```

Internals: This program creates a $$$.SUB file on drive A:, erasing any existing $$$.SUB file. The lines present in the command string are copied to the $$$.SUB file, for execution in the typed order.

Notes:

- o The command processor (CCP) is to be loaded automatically by CP/M on exit.
- o To return to the calling program, the string passed to the CCP must contain a re-load for the current program.
- o CP/M is not concurrent. There is no efficient way to save the current process for a restart.

Other ways to do the same thing:

- o Use run(). It accepts a CP/M command line to load and run a COM file. Only one line.
- o See also chain().

The tan Function

Purpose:

Compute the tangent of real float value x. The value x is assumed in radians (not degrees).

Function Header:

```
float tan(x)
float x;            Float value in range -8*PI to +8*PI,
                    PI = 3.14159.
```

Returns:

number	Answer between -INF and +INF, errno = 0.
INF	x approx (2n+1)PI/2 error, errno = ERANGE.
ZERO	Argument too large negative or positive (greater than 8*PI), errno = EDOM.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float tan();
     errno = 0;
     printf("Tan(x) accepts x from -25.13 to +25.13 radians\n");
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("tan(%f) = %f, errno = %d\n",x,tan(x),errno);
}
```

Notes:

- o Use deg() and rad() for conversions.

- o The method used is sin(x)/cos(x) in range, or rational approximation, Cheney & Hart, in the critical range:

```
tan(x) = sin(x)/cos(x)  for x not within .01 of
                        a multiple of PI/2
tan(x) = new approx     otherwise
```

- o Current problems:

  First, if fabs(x) > 25.13, then we probably can't compute it at all. The number 25.13 comes from

  8*PI = 25.13274123 (12-digits)
  8*3.14159 = 25.13272 (6-digits)
  error = -2.12288e-05
  So to maintain 6-digit accuracy restrains
  computation to -8PI < x < 8PI.

  Let us assume 0 <= x <= pi/2. Let u = pi/2 - x. If u is too close to zero, then computation fails:

```
sin(x)/cos(x) = sin(x)/cos(pi/2-u)
              = sin(x)/sin(u)
              = cos(u)/sin(u)
              = 1./u approximately
```

  This is cured by using a different rational function near pi/2, namely a Taylor expansion:

  $$\tan x = (1/u)(1 - u^2/2 + u^4/24)/(1 - u^2/6 + u^4/120)$$

  At u = .009999, tan x = 100.006668, but 1/u = 100.010001. For 0 <= x < BADTAN, sin(x)/cos(x) is a good approximation. It looks like we need the above rational, but perhaps a better choice of BADTAN would allow use of 1/u.

- o We define BADTAN = 1.560796327, which is pi/2 - .01 or 89.427 degrees. This rather arbitrary value comes from comparison with other tangent functions on micros and mainframes.

The tanh Function

Purpose:

Compute the hyperbolic tangent of real float value x.

Function Header:

float tanh(x)	Hyperbolic tangent $(e^{x} - e^{-x})/(e^{x} +e^{-x})$.
float x;	Positive, negative or zero argument.

Returns:

number	x in range
INF	x too large positive (see notes).
-INF	x too large negative (see notes).
	errno = ERANGE returned to flag error.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float tanh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("tanh(%f) = %f, errno = %d\n",x,tanh(x),errno);
}
```

Notes:

- o Method used is tanh(x) = (exp(2x) - 1)/(exp(2x) + 1).

- o The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The timer Function

```
========================================================================
```

Purpose:

Time an event under CP/M with 2ms clock at address 11. Works under CP/M 2.203 H89 or CP/M-85 Z100. Adaptable to most systems.

Function Header:

```
VOID timer(n)
int  n;                 0 to start timer
                        1 to print out the elapsed time
```

Returns:

Nothing of interest

Example: Time an internal loop and report the elapsed time.

```
#include <stdio.h>
main()
{
int i;
     timer(0);
     for(i=0;i<30000;++i) {
     /* an empty loop. The Sieve Benchmark would be interesting */
     }
     timer(1);
}
```

The timer routine prints the elapsed time using integer arithmetic and printf. It is accurate to 1/10 of a second.

Notes:

- o Uses a fixed CLOCK word at address 11. The expected 2ms clock is present only on the H89 under CP/M and the Z100 under CP/M-85.

- o See the source in TIMER.C for other ideas.

The toascii Function

==

Purpose:

Converts an integer into the ASCII range 0 to 127 by stripping the parity bit.

Function Header:

```
     int toascii(x)
     int x;                  Integer to convert to ASCII range.
```

Returns:

The converted value, in the range 0 to 127 decimal.
Logically equivalent to (x & 127).

Example: Read a word processor file such as produced by MicroPro's Wordstar, strip the file to ASCII range, and print on the printer.

```
     #include <stdio.h>
     main(argc,argv) int argc; char **argv;
     {
     int x;
     FILE *fp,*fo,*fopen();

          if(argc <= 1) {
               puts("Usage: A>strip inputfile"); exit();
          }
          if((fp = fopen(argv[1],"r")) == (FILE *)0) {
               puts("Bad open of input file"); exit();
          }
          if((fo = fopen("LST:","w")) == (FILE *)0) {
               puts("Printer failure"); exit();
          }
          while((x = getc(fp)) != EOF) {
               putc(toascii(x),fo);
          }
          fclose(fo); fclose(fp);
     }
```

Notes:

- With IBM data using the EBCDIC standard character set, the ASCII standard toascii() is totally useless.
- Toascii() is an honest function and not a macro. There are no side effects.

The toint Function

===

Purpose:

Converts a hex digit into a integer in the range 0 to 15. For example, toint('A') = 10, toint('f') = 15, toint('0') = 0.

Function Header:

```
     int toint(x)
     char x;                    Character to convert
```

Returns:

n	In range 0...15 for a valid upper or lower case hexadecimal digit from the character set 0123456789ABCDEFabcdef
-1	Error, an invalid hex digit.

Example: Read and convert hex digits from stdin.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int x,y; FILE *fp,*fopenb();
  if(argc > 1) {
  if(fp = fopenb(argv[1],"w")) {
    while((x = getchar()) != EOF) {
       if(isspace(x)) continue;
       if((y = getchar()) != EOF)
            putc(toint(x)*16+toint(y),fp);
    }
    fclose(fp);
  }
}
```

Notes:

- o Implementations of C that use code macros often put the function toint() into the CTYPE.H file. Here is the standard toint() code:

```
int toint(x) char x;
{
  if(isdigit(x)) return (x - '0');
  x = toupper(x) - 'A' + 10;
  return (10 <= x && x <= 15) ? x : -1;
}
```

The tokens Function

Purpose:

Decodes a CP/M command line into token strings. Processes redirection files < and >. Handles strings with imbedded white space with single or double quote delimiters.

Function Header:

```
int tokens(table,cmdtail)
char *table[];              Pointer table to be filled.
char *cmdtail;              Special buffer area. The first byte is
                            the length of the string. Following are
```

The token table is altered by the function call as follows:

table[0]:	address of '<' redirection file name
table[1]:	address of '>' redirection file name
table[2]:	address of the first token
.	
.	
.	
table[n]:	address of the last token
table[n+1]:	always set to -1, as a marker

Returns:

The number of tokens actually processed.

Example: Do our own token decoding in lowercase.

```
#include <stdio.h>
decodtokens(table)
char *table[];
{
int n,i;
char s[129];
     printf("Enter a line: ");
     gets(&s[1]);
     s[0] = strlen(&s[1]);
     table[0] = table[1] = "NO DATA";
     n = tokens(table,s);
     printf("table[0] = Re-direction <: %s\n",table[0]);
     printf("table[1] = Re-direction >: %s\n",table[1]);
     for(i=2;i<2+n;++i) printf("table[%d] = %s\n",table[i]);
}
```

Notes:

- o The character pointer **source** points to a buffer area in memory which contains the command line tail. This area is disected into tokens, which are delimited by nulls, and the addresses are consecutively stored in the table.

- o Any redirection commands cause the entries **table[0]** and **table[1]** to be updated, otherwise these memory locations are left in entry state. Programs which use tokens() must initialize table[0] and table[1] (to a null string).

- o Decoding recognizes both double and single quotes as commands to stop white space searching (until the quote is repeated). Therefore, tokens may be passed which contain white space.

- o Both upper and lower case characters are recognized by tokens(). However, the CCP will convert lowercase to uppercase. Programs which require lower case input can therefore invoke tokens() inside the program, and service command line strings as usual. **See also insert(), chain(), run().**

- o The redirection file names are obtained from pointers table[0] and table[1]. This feature allows internal redirection changes, by closing streams **stdin** and **stdout,** then re-opening these streams with the supplied file names. Implement this by writing your own **Copen_()** and **Cexit_().** See also UNIX.H and the function freopen().

The tolower Function

===

Purpose:

Converts a character in the range A-Z to lower case.

Function Header:

```
int tolower(x)
int x;                          Character to convert.
```

Returns:

The character x, if no conversion was required.
The lowercase equivalent of x, if 'A' <= x <= 'Z'.

Example: Convert an input line to all lowercase.

```
#include <stdio.h>
main()
{
char s[129];
int i;

     printf("Enter a line of text: ");
     gets(s);
     for(i=0;i<strlen(s);++i) s[i] = tolower(s[i]);
     printf("Lowercase line:\n%s\n",s);
}
```

Notes:

- o On some machines this function is a macro. Beware of side effects when porting code. If you write it, then try to do so without introducing possible side effects (like tolower(*p++)).
- o Tolower() is an honest function, it is not a macro, and it has no side effects.
- o Some macro implementations do not bother testing to see if the integer x to be converted is in the range A-Z. The net result is an invisible bug. Look in STDIO.H on the target for such nightmares.

The toupper Function

```
===========================================================================
```

Purpose:

Converts a character in the range a-z to upper case A-Z.

Function Header:

```
int toupper(x)
int x;                        Character to convert.
```

Returns:

The character x, if no conversion was required.
The uppercase equivalent of x, if 'a' <= x <= 'z'.

Example: Convert an input line to all uppercase.

```
#include <stdio.h>
main()
{
char s[129];
int i;

     printf("Enter a line of text: ");
     gets(s);
     for(i=0;i<strlen(s);++i) s[i] = toupper(s[i]);
     printf("UPPERcase line:\n%s\n",s);
}
```

Notes:

- o On some machines this function is a macro. Beware of side effects when porting code. If you write it, then try to do so without introducing possible side effects (like toupper(*p++)).

- o Toupper() is an honest function, it is not a macro, and it has no side effects.

- o Some macro implementations do not bother testing to see if the integer x to be converted is in the range a-z. The net result is an invisible bug. Look in STDIO.H for such problems.

- o See also cvupper() for a function that converts an entire line to upper case with a single function call.

The ttyname Function

===

Purpose:

Get a device name string pointer from an open file descriptor.

Function Header:

```
char *ttyname(fd)
int  fd;                 File descriptor for an open file, use
                         fd = fileno(fp) for a stream FILE *fp.
```

Returns:

(char *)p	Pointer to static memory containing the string for the device name.
(char *)0	Error, file not open or was not a valid device.

Example: Print the name of the device attached to stream fp.

```
#include <unix.h>
#include <stdio.h>
whatdevice(fp)
FILE *fp;
{
char *p,*ttyname();
     p = ttyname(fileno(fp));
     if(p == (char *)0) puts("Not a device");
     else puts(p);
}
```

Notes:

- o Valid devices are con:, rdr:, pun:, lst:
- o For Unix compatibility, assume this function returns only the name of the controlling terminal.
- o The source code switches on device numbers as follows:

```
switch(fd) {
case   0:
case 252: return "CON:";
case 253: return "RDR:";
case 254: return "PUN:";
case 255: return "LST:";
default:  return 0;
}
```

The ungetc Function

Purpose:

Pushes a character back onto an open stream, ready to be used by the next call to getc(). Only a one-character push-back is allowed.

Function Header:

```
int ungetc(x,fp)
int x;                       Character to be pushed back.
FILE *fp;                    Open stream pointer.
```

The value of x is ignored for a disk file, but it must be present in the argument list. For the console stream, the character x is placed in the console buffer, if possible (it could be full). An EOF character cannot be pushed back, and it is an error to attempt this feat.

Returns:

EOF	Push-back failed
x	Push-back worked (or x = EOF)

Example: Push a string back onto the console input buffer.

```
#include <stdio.h>
main()
{
char s[129],t[129];
int i;
     printf("Enter a string: ");
     gets(s);
     for(i=0;i<128;++i) {
          if(s[i]) ungetc(s[i],stdin); else break;
     }
     ungetc('\n',stdin);
     gets(t);
     printf("Here's t: %s\n",t);
}
```

Example: Read an input file until % is found, then print the line on which it occurs.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
FILE *fp,*fopen();
int i;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) exit(0);
     while((i = getc(fp)) != EOF && i != '%') ;
     if(i == EOF) {
          printf("Flag %% not found\n"); exit(0);
     }
     else {
          printf("Found %%\n");
          ungetc(i,fp);
          fgets(s,128,fp);
          printf("%s",s);
     }
     fclose(fp);
}
```

Notes:

- o Calls to fseek() or freopen() or seek() are unaffected by a call to ungetc(), because data in the file buffer is not changed.
- o Calls to ungetc() for disk files may have the effect of moving the file pointer back to a previous buffer, in which case the action taken is to reload the file buffer from disk.
- o The console limit is 136 characters, and you may attempt to push back an entire string to see how this works. Under re-direction this process will fail to have the desired result because the push-back for files does not fool with the data on disk.
- o Crazy code that stuffs the file buffers with data that was not on disk is not acceptable. It will have to be re-written. Sane code will compile and run normally.
- o Scanf() uses ungetc(). If your code mixes fseek, ungetc, scanf, then it may not be portable, even though it executes properly with this library.

The unix I/O and system Functions

Purpose:

Allow UNIX System III code to compile under C/80. Stubs are in the file UNIXIO.H, ready to edit into your program. A copy of all stubs is provided in CLIBU.REL, for convenience. See UNIXIO.H for more details.

Reference texts:

Basic Reference: Banahan & Rutter, The Unix Book, Wiley Press,
New York (1983).

Unix Reference: The Unix Programmers Manual.

Function Headers:

=====UNIX SYSTEM CALLS======

```
int acct(file)                    Return 0.
char *file;

unsigned alarm(seconds)           Return 0.
unsigned second;

char *cuserid(s)                  Return (char *)0.
char *s;

int dup(old)                      Return -1 for error.

int dup2(old,new)                 Return -1 for error.

int fork()                        Return -1 for error.

#include <sys/types.h>
#include <sys/stat.h>
int fstat(fd,buf)                 Return -1 for error.
int fd;
struct stat *buf;

#include <sys/types.h>
#include <sys/timeb.h>
void ftime(tp)                    Void return.
struct timeb *tp;

int getpid()                      Return 0.

int getuid()                      Return 0.

int getgid()                      Return 0.
int geteuid()                      Return 0.

int getegid()                      Return 0.

int ioctl(fd,request,argp)         Return -1 for error.
int fd,request;
struct sgtty *argp;

int getpgrp()                      Return 0.

int getppid()                      Return 0.

int kill(pid,sig)                  Return -1 for error.
int pid,sig;

int link(path1,path2)              Return -1 for error.
char *path1,*path2;

int mknod(path,mode,dev)           Return -1 for error.
char *path;
int mode,dev;

int mount(spec,dir,rwflag)         Return -1 for error.
char *spec,*dir;
int rwflag;

int nice(incr)                     Return 0.
int incr;

void pause()                       Void return.

int pclose(fp)                     Return -1 for error.
FILE *fp;

int pipe(fd)                       Return -1 for error.
int fd[2];

FILE *popen(command,type)          Return 0.
char *command,*type;

void profil(buff,bufsize,offset,scale)        Void return.
char *buff;
int bufsize,offset,scale;

int ptrace(request,pid,addr,data)  Return -1 for error.
int request,pid,addr,data;

int putpwent(p,fp)                 Return -1 for error.
struct passwd *p;
FILE *fp;

int setpgrp()                      Return 0.
int setgid(gid)                    Return -1 for error.
int gid;

int setuid(uid)                    Return -1 for error.
int uid;

#include <sys/types.h>
#include <sys/stat.h>
int stat(name,buf)                 Return -1 for error.
char *name;
struct stat *buf;

void sync()                        Void return.

long time(tloc)                    Returns (long)0
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
long times(buffer)                 Return -1 for error.
struct tbuffer *buffer;

int stime(tp)                      Return -1 for error.
long *tp;

int umask(cmask)                   Return 0.
int cmask;

int umount(spec)                   Return -1 for error.
char *spec;

int uname(name)                    Return 0.
struct utsname *name;

char *userid(s)                    Return (char *)0
char *s;

int ustat(dev,buf)                 Return -1 for error.
int dev;
struct ustat *buf;

#include <sys/types.h>
int utime(path,times)              Return -1 for error.
char *path;
struct utimbuf *times;  or  time_t timep[2];

int wait(stat_loc)                 Return -1 for error.
int *stat_loc;
```

=====UNIX MISCELLANY=====

```
    #include <grp.h>
    struct group *getgrent()              Return 0.

    #include <grp.h>
    struct group *getgrgid(gid)           Return 0.
    int gid;

    #include <grp.h>
    struct group *getgrnam(name)          Return 0.
    char *name;

    int setgrent()                        Return 0.

    int endgrent()                        Return 0.

    char *getlogin()                      Return (char *)0.

    int getpw(uid,buf)                    Return -1 for error.
    int uid;
    char *buf;

    #include <pwd.h>
    struct passwd *getpwuid(uid)          Return 0.
    int uid;

    #include <pwd.h>
    struct passwd *getpwent()             Return 0.

    #include <pwd.h>
    struct passwd *getpwnam(name)         Return 0.
    char *name;

    int setpwent()                        Return 0.

    int endpwent()                        Return 0.

    int monitor(lowpc,highpc,buffer,bufsize,nfunc)     Return 0.
    int (*lowpc)();
    int (*highpc)();
    short buffer[];
    int bufsize;
    int nfunc;

    int sleep(seconds)                    Return 0.
    unsigned seconds;
```

The unlink Function

Purpose:

Erase a disk file from the disk directory.

Function Header:

```
int unlink(filename)
char *filename;                    Existing file name.
```

Returns:

```
 0            No error.
-1            Error, operation failed.
```

Example: Delete a disk file.

```
#include <stdio.h>
main()
{
int unlink();
char s[129];
       printf("File name to erase: ");
       gets(s);
       if(unlink(s) == -1) puts("Erase failed");
}
```

Example: Delete a list of files.

```
#include <stdio.h>
main()
{
int unlink();
  if(unlink("A:*.COM") == -1) puts("Erase failed");
}
```

Notes:

o Unlink() is a Bell Labs standard function.

o The function accepts a wildcard argument. You may delete many files at once with a single call to unlink().

The utoa, utoax, utoao, utoab Functions

Purpose:

Convert 16-bit integer value to printable characters. String after conversion is between 5 and 17 bytes in length. See below.

Function Header:

```
int utoa(n,s)       Unsigned integer to decimal digits.
int n;              Integer to convert.
char s[6];          Storage area for the string.

int utoax(n,s)      Unsigned integer to hexadecimal digits.
int n;              Integer to convert.
char s[5];          Storage area for the string.

int utoao(n,s)      Unsigned integer to octal digits.
int n;              Integer to convert.
char s[7];          Storage area for the string.

int utoab(n,s)      Unsigned integer to binary digits.
int n;              Integer to convert.
char s[17];         Storage area for the string.
```

Returns:

The number of digits converted. Equals strlen(s).

Example: Print integer conversions.

```
#include <stdio.h>
main()
{
int i; char s[129];
     do {
          type("Enter number: ");
          gets(s);
          i = atoi(s);
          itoa(i,s); puts(s);
          utoa(i,s); puts(s);
          utoax(i,s); puts(s);
          utoao(i,s); puts(s);
          utoab(i,s); puts(s);
     } while(i);
}
```

Notes:

- o All routines jump to the assembler interface $STRCV after loading the accumulator. Same routine as used by printf.

The waitz Function

Purpose:

Waits for a specified number of milliseconds.

Function Header:

```
VOID waitz(n)
int n;              Wait for 2n milliseconds. Uses the 2
                    millisecond memory clock of the Z90
                    and Z100 computers.
```

Returns:

Nothing. CPU is bound to a loop until timeout.

Example: Wait for time entered on the command line.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv[];
{
int i,j;
  if(argc<=1) exit();
  j = atoi(argv[1]); if(j+j==0) exit();
  for(i=0;i<10;++i) {
  printf("Waiting for %dms..\n",j+j);
  timer(0); waitz(j); timer(1);
  }
}
```

Notes:

- If your machine does not have a memory-based timer, then look around for an 8253 timer chip or similar timing device.

- The timer interface is used because of accuracy. You can build a software timer that does essentially the same thing, but it will be sensitive to CPU speed.

- Works on the H89, Z90, Z100 with CP/M-85. Very little chance on other systems. See the source code for ideas on how to implement it on other computers.

- Attempts to test this function on machines that are not hardware equipped as cited above will result in a machine hang. The software loop is an infinite loop in case the timer does not change (unused location 11).

The wrDISK Function

Purpose:

Direct-disk write function for CP/M 2.2.

Function Header:

```
wrDISK(track,record,drive,buffer)

int track;               Physical track to write, 0 = first.
int record;              Record to write on track, 1 = first.
int drive;               Drive number. Use 0 for A:, 1 for B:, etc.
char buffer[128];        Location of data to write to disk.
```

Returns:

Buffer written on success.
Error code for a disk write is returned. Should be 0 for success.

Example: Write track 0 sector 1, the Zenith label sector.

```
#define libl CLIBX
#include <stdio.h>
main()
{
int i;
char buf[128];
#define TRK 0
#define SEC 1
#define DSK 0

    fillchr(buf,'\0',128);
    printf("wrDISK() = %d\n", i = wrDISK(TRK,SEC,DSK,buf));
    for(i=0;i<128;++i) printf("%02x\n",buf[i]&255);
}
```

Notes:

o Note that wrDISK() is coded in assembly language and is highly non-portable. The function wrDISK() can be written in terms of the function bios(). The latter is recommended for portability, especially to CP/M-68K and Unix. The portable source:

```
bios(12,buffer);                Under CP/M-68K the second
bios(9,drive);                  argument of bios() has a
bios(10,track);                 cast of (long).
bios(11,record);                Under CP/M-80 2.2, the cast
return bios(14,0);              is (int).
```

o Immediate writes to disk must use bios(14,1). This problem occurs with LRU buffering (Z100, for example). Normal writes use bios(14,0) with possibly delayed physical write.

write_ - C/80 Standard write

The write Function for C/80
===

Purpose:

Writes a block of characters to the disk in binary mode. The amount written must be a multiple of 128. Adheres to the Software Toolworks standard for the write() function. Use UNIX.H to obtain the Bell Labs Unix standard.

Function Header:

```
#include <stdio.h>
unsigned write_(fp,buffer,count)
FILE *fp;                           Open stream pointer.
char *buffer;                       Buffer which contains data.
unsigned count;                     Number of bytes to write.
                                    This number must be a multiple
                                    of 128.
```

Returns:

The number of bytes actually written, which should equal the number count above. It is an error if write_() != count.

Example: Write 2048 bytes to a disk file.

```
#include <stdio.h>
put2048(fp,buf)
FILE *fp;
char buf[2048];
{
unsigned x,write_();
     x = write_(fp,buf,2048);
     if(x != 2048) puts("Write failure");
}
```

Notes:

- Streams with descriptors 252,253,254,255 are devices. Write () under C/80 cannot use these device descriptors. Use putc() for safety.

- A write failure is usually due to end of media. Since the C/80 write_() function operates in binary mode only, there is no carriage return and linefeed translation. Nor does the function insert ctrl-Z at the end of the file.

- The write_() function under C/80 operates in quanta of 128 bytes. You cannot write 1 character. See UNIX.H and the Unix-style write() and fwrite() functions for an alternative.

- This write_ function uses CP/M function 34 with manual advance of the record number. The initial record number is saved in the system variable IOsect[fp].

- The symbol **write_** is the same label as **write** in CLIB.REL. Portable source code should use **write_** for the C/80 write(). Other systems will use read() and write() as in UNIX.H.

- Old C/80 code may be compiled without changes, because **write** is a global in CLIB.REL which has the correct meaning for such source code. New source code should use **write_**.

- A file opened for update "u" may be read to the end with the **read_()** function followed by rewind(fp) and access by the **write_()** function. Mixed calls of read_/write_/getc/putc will probably fail.

The write and writea Functions

===

Purpose:

Transfers bytes from main memory to a non-stream file. Move n bytes per call from a preset buffer. The transfer suffers from cr/lf translation if the file was opened in Ascii mode (the usual case).

Function Header:

```
int  writea(fd,buffer,n)
int  fd;                 File descriptor, fd=fileno(fp) where
                         fp is the stream pointer.
char *buffer;            Pointer to base of memory storage which
                         contains the data to be written to disk.
int  n;                  Number of bytes to transfer, 0...32767.
```

Returns:

```
 m      The number of bytes actually transferred, an integer
        quantity 0...32767. Could be different from n.
-1      Error
```

WARNING: In UNIX.H appears the definition **#define write writea.**

Example: Write 1 byte to a file.

```
#include <unix.h>
#include <stdio.h>
main()
{
FILE *fp,*fopen();
int x;
     fp = fopen("TMP","w");
     x = 'A';
     write(fileno(fp),&x,1);
     fclose(fp);
}
```

Notes:

- o Note that UNIX.H installs a new name for write(). You actually use writea(). Beware of the name conflict with C/80 write(). The primitive C/80 write() is used implicitly. All Unix re-direction is in force, because of the explicit use of putc().

- o Useful for writing out Ascii data from the current file pointer.

- o This function differs from the C/80 primitive write() in that quanta of 128 bytes are not required and in addition you can write to a device. Devices CON:, PUN:, LST: are recognized, but CR/LF translation occurs.

The writeb Function
==

Purpose:

Transfers bytes from main memory to a file opened by descriptor. Move n bytes per call from a preset buffer. The transfer is done in binary mode regardless of how the file was opened.

Function Header:

```
     int  writeb(fd,buffer,n)
     int  fd;                 File descriptor, fd=fileno(fp) where
                              fp is the stream pointer.
     char *buffer;            Pointer to base of memory storage which
                              contains the data to be written to disk.
     int  n;                  Number of bytes to transfer, 0...32767.
```

Returns:

```
     m       The number of bytes actually transferred, an integer
             quantity 0...32767. Could be different from n.
    -1       Error
```

WARNING: In UNIX.H appears the definition

```
               #define write writea
```

Example: Write a linefeed only to an output file opened in ASCII mode.

```
     #include <unix.h>
     #include <stdio.h>
     main()
     {
     FILE *fp,*fopen();
     int x;
          fp = fopen("TMP","w");
          x = '\n';
          writeb(fileno(fp),&x,1);
          fclose(fp);
     }
```

Example: Write a linefeed only to the console.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x;
     x = '\n';
     printf("Watch it happen");
     writeb(fileno(stdout),&x,1);
     printf("here");
}
```

Notes:

- o The header file UNIX.H names write() as writea(), an ASCII access. Beware of the name conflict write() creates with the C/80 library.

- o Useful for writing binary data from the current file pointer.

- o The console device is treated specially so that a linefeed is not converted to CR/LF. For other devices like the printer, bytes are output just as they appear in the buffer without further translation.

- o For disk files, the mode of the file is temporarily changed to binary and the bytes are output from the buffer to the stream buffer. Recall that under the present library, all disk files are stream files. However, the fill character used for the stream and the end-of-file mark are not affected.

- o This function differs from the C/80 primitive write() in that quanta of 128 bytes are not required and in addition you can write to a device.