11 Additional Utility Programs

This chapter contains the narrated source code for several useful utility programs. Two groups of such programs are included—those that supplement Digital Research's standard utility programs, and those that work in conjunction with features shown in the enhanced BIOS (Figure 8-10).

To avoid unnecessary detail, the programs shown in this chapter are all written in the C language. C is a good language to use for such purposes since it can show the overall logic of a program without the clutter of details common in assembly language.

In order to reuse as much source code as possible, this chapter includes a "library" of all the general-purpose C functions that can be called from within any of the utility programs. This file, called "LIBRARY.C", is shown in Figure 11-1. Once a utility program has been compiled, the necessary functions from the library can be linked with the utility's binary output to form the ".COM" file.

```
/* Library of commonly-used functions */

#include <LIBRARY.H>     /*  Standard defines and structures */

/*      Configuration block access      */

/*=============================================================*/
char
*get_cba(code)          /* Get configuration block address */
/*=============================================================*/
/* This function makes a call to a "private" entry in the BIOS
   jump vector to return the address of a specific data object in
   the BIOS. The code indicates which object is required.
   Each program using this function could make a direct call to
   the BIOS using the biosh() function provided by BDS C. This
   function provides a common point to which debugging code can
   be added to display the addresses returned. */

/* Entry parameters */                                              a
int code;       /* Code that specifies the object
                        whose address is required */
/* Exit Parameters
   Address returned by the BIOS routine */

{
char *retval;           /* Value returned by the BIOS */

        retval = biosh(CBGADDR,code);
 /* printf("\nget_cba : code %d address %4x",code,retval); */
        return retval;
} /* End of get_cba(code) */



/*      Character manipulation functions        */


/*=============================================================*/
strscn(string,key)              /* String scan */
/*=============================================================*/
/* This function scans a 00-terminated character string looking
   for a key string in it. If the key string is found within the
   string, the function returns a pointer to it. Otherwise it
   returns a value of zero. */
/* Entry parameters */
char *string;           /* String to be searched */
char *key;              /* Key string to be searched for */

/* Exit parameters
   Pointer to key string within searched string, or
   zero if key not found
*/                                                                  b

{
while (*string)         /* For all non-null chars. in string */
        {
        if ((*string == *key) &&        /* First char. matches */
            (sstrcmp(string,key) == 0)  /* Perform substring
                                           compare on rest */
           )
                return string;          /* Substring matches,
                                           return pointer */
        string++;                       /* Move to next char. in string */
        }
return 0;                               /* Indicate no match found */
} /* End of strscn */

/*=============================================================*/
ustrcmp(string1,string2)        /* Uppercase string compare */
/*=============================================================*/
/* This function is similar to the normal strcmp function;          c
   it differs only in that the characters are compared as if they
   were all uppercase characters -- the strings are left
   unaltered. */
```

Figure 11-1. LIBRARY.C, commonly used functions, in C language

```
/* Entry Parameters */
char *string1;          /* Pointer to first string */
char *string2;          /* Pointer to second string */

/* Exit parameters
   0 - if string 1 = string 2
   -ve integer if string 1 > string 2
   +ve integer if string 1 < string 2
*/

{
int count;              /* Used to access chars. in both strings */
                                                                        c
count = 0;              /* Start with the first character of both */

        /* While string 1 characters are non-null, and
           match their counterparts in string 2. */
while (string1[count] == string2[count])
        {
        if (string1[++count] == '\0')   /* Last char. in string 1 */
                return 0;               /* Indicate equality */
        }
return string2[count] - string1[count]; /* "Compare" chars. */

} /* End of sstrcmp */


/*==============================================================*/
sstrcmp(string,substring)               /* Substring compare */
/*==============================================================*/
/* This function compares two strings. The first, string, need not
   be 00-terminated. The second, substring, must be 00-terminated.
   It is similar to the standard function strcmp, except that the
   length of the substring controls how many characters are compared. */

/* Entry parameters */
char *string;           /* Pointer to main string */
char *substring;        /* Pointer to substring */

/* Exit parameters
   0 - substring matches corresponding characters in string
   -ve integer if char. in string is > char. in substring
   +ve integer if char. in string is < char. in substring
*/                                                                      d

{
int count;      /* Used to access chars. in string and substring */

count = 0;      /* Start with the first character of each */

        /* While substring characters are non-null, and
           match their counterparts in string. */
while (string[count] == substring[count])
        {
        if (substring[++count] == '\0') /* Last char in substring */
                return 0;               /* Indicate equality */
        }
return substring[count] - string[count];        /* "Compare" chars. */

} /* End of sstrcmp */


/*==============================================================*/
usstrcmp(string,substring)      /* Uppercase substring compare */
/*==============================================================*/
/* This function compares two strings. The first, string, need not
   be 00-terminated. The second, substring, must be 00-terminated.
   It is similar to the substring compare above except all
   characters are made uppercase. */                                    e

/* Entry parameters */
char *string;           /* Pointer to main string */
char *substring;        /* Pointer to substring */

/* Exit parameters
   0 -- substring matches corresponding characters in string
```

Figure 11-1. (Continued)

```
   -ve integer if char. in string is > char. in substring
   +ve integer if char. in string is < char. in substring
*/
{
int count;       /* Used to access chars in string and substring */

count = 0;       /* Start with the first character of each */

        /* While substring characters are non-null, and
           match their counterparts in string. */
while (toupper(string[count]) == toupper(substring[count]))
        {
        if (substring[++count] == '\0') /* Last char. in substring */
                return 0;               /* Indicate equality */
        }
return substring[count] - string[count];        /* "Compare" chars. */
} /* End of usstrcmp */                                                    e


/*==============================================================*/
comp_fname(scb,name)            /* Compare file names */
/*==============================================================*/
/* This function compares a possibly ambiguous file name
   to the name in the specified character string. The number of
   bytes compared is determined by the number of characters in
   the mask.
   This function can be used to compare file names and types,
   or, by appending an extra byte to the mask, the file names,
   types, and extent numbers.
   For file directory entries, an extra byte can be prefixed to
   the mask and the function used to compare user number, file
   name, type, and extent.
   Note that a "?" in the first character of the mask will NOT
   match with a value of 0xE5 (this value is used to indicate
   an inactive directory entry). */

/* Entry parameters */
struct _scb *scb;       /* Pointer to search control block */
char *name;             /* Pointer to file name */

/* Exit parameter
   NAME_EQ if the names match the mask
   NAME_LT if the name is less than the mask
   NAME_GT if the name is greater than the mask
   NAME_NE if the name is not equal to the mask (but the outcome
        is ambiguous because of the wildcards in the mask)
*/                                                                        f

{
int count;              /* Count of the number of chars. processed */
short ambiguous;        /* NZ when the mask is ambiguous */
char *mask;             /* Pointer to bytes at front of SCB */

/* Set pointer to characters at beginning of search control block */
mask = scb;

        /* Ambiguous match on user number, matches
           only users 0 - 15, and not inactive entries */
if (mask[0] == '?')
        {
        if (name[0] == 0xE5)
                return NAME_NE; /* Indicate inequality */
        }
else    /* First char. of mask is not "?" */
        {
        if (mask[0] != name[0]) /* User numbers do not match */
                return NAME_NE; /* Indicate inequality */
        }

/* No, check the name (and, if the length is such, the extent) */
for (count = 1;                 /* Start with first name character */
     count <= scb -> scb_length; /* For all required characters */
     count++)                   /* Move to next character */
        {
        if (mask[count] == '?') /* Wildcard character in mask */
```

Figure 11-1. (Continued)

```
                {
                ambiguous = 1;  /* Indicate ambiguous name in mask */
                continue;       /* Do not make any comparisons */
                }
        if (mask[count] != (name[count] & 0x7F))
                {               /* Mask char. not equal to FCB char. */
                if (ambiguous)  /* If previous wildcard, indicate NE */
                        return NAME_NE;
                else
                        /* Compare chars. to determine relationship */
                        return (mask[count] > name[count] ?                 f
                                NAME_LT : NAME_GT);
                }
        }
        /* If control reaches here, then all characters of the
        mask and name have been processed, and either there
        were wildcards in the mask, or they all matched. */
return NAME_EQ;         /* Indicate mask and name are "equal" */

} /* End of comp_fname */


/*==============================================================*/
conv_fname(fcb,fn)              /* Convert file name for output */
/*==============================================================*/
/* This function converts the contents of a file control
   block into a printable string "D:FILENAME.TYP." */

/* Entry parameters */
struct _fcb *fcb;               /* Pointer to file control block */
char *fn;                       /* Pointer to area to receive name */

{
        /* If the disk specification in the
           FCB is 0, use the current disk */
*fn++ = (fcb -> fcb_disk) ? (fcb -> fcb_disk + ('A'-1)) :
        (bdos(GETDISK) + 'A');                                              g

*fn++ = ':';                            /* Insert disk id. delimiter */

movmem(&fcb -> fcb_fname,fn,8);         /* Move file name  */
fn += 8;                                /* Update pointer */
*fn++ = '.';                            /* Insert file name/type delimiter */
movmem(&fcb -> fcb_fname+8,fn,3);       /* Move file type */
*fn++ &= 0x7F;                          /* Remove any attribute bits */
*fn++ &= 0x7F;                          /* Remove any attribute bits */
*fn++ &= 0x7F;                          /* Remove any attribute bits */
*fn = '\0';                             /* Terminator */

} /*  End of conv_fname  */


/*==============================================================*/
conv_dfname(disk,dir,fn)        /* Convert directory file name for output */
/*==============================================================*/
/* This function converts the contents of a file directory entry
   block into a printable string "D:FILENAME.TYP," */

/* Entry parameters */
short disk;                     /* Disk id. (A = 0, B = 1) */
struct _dir *dir;               /* Pointer to file control block */
char *fn;                       /* Pointer to area to receive name */

{
        /* Convert user number and disk id. */                              h
sprintf(fn,"%2d/%c:",dir -> de_userno,disk + 'A');
fn += 5;                        /* Update pointer to file name */

movmem(&dir -> de_fname,fn,8);  /* Move file name  */
fn += 8;                        /* Update pointer */
*fn++ = '.';                    /* Insert file name/type delimiter */

movmem(&dir -> de_fname+8,fn,3); /* Move file type */
*fn++ &= 0x7F;                  /* Remove any attribute bits */
*fn++ &= 0x7F;                  /* Remove any attribute bits */
*fn++ &= 0x7F;                  /* Remove any attribute bits */
*fn = '\0';                     /* Terminator */
```

Figure 11-1. (Continued)

```
} /*  End of conv_dfname  */                                          h


/*===========================================================*/
get_nfn(amb_fname,next_fname)   /* Get next file name */
/*===========================================================*/
/* This function sets the FCB at "next_fname" to contain the
   directory entry found that matches the ambiguous file name
   in "amb_fname."
   On the first entry for a given file name, the most significant
   bit in the FCB's disk field must be set to one (this causes a
   search first BDOS call to be made). */

/* Entry parameters */
struct _fcb *amb_fname; /* Ambiguous file name */
struct _fcb *next_fname;/* First byte must have ms bit set for
                           first time entry)*/

/* Exit parameters
   0 = No further name found
   1 = Further name found (and set up in next_fname)
*/

{
char bdos_func;          /* Set to either search first or next */
char *pfname;            /* Pointer to file name in directory entry */

        /* Initialize tail-end of next file FCB to zero */
setmem(&next_fname -> fcb_extent,FCBSIZE-12,0);

bdos_func = SEARCHF;     /* Assume a search first must be given */     i

if (!(next_fname -> fcb_disk & 0x80))    /* If not first time */
        {
                /* search first on previous name */
        srch_file(next_fname,SEARCHF);
        bdos_func = SEARCHN;            /* Then do a search next */
        }
else    /* First time */
        next_fname -> fcb_disk &= 0x7F; /* Reset first-time flag */

        /* Refresh next_fname from ambiguous file name
           (move disk, name, type) */
movmem(amb_fname,next_fname,12);

        /* If first time, issue search first, otherwise
           issue a search next call. "srch_file" returns
           a pointer to the directory entry that matches
           the ambiguous file name, or 0 if no match */
if (!(pfname = srch_file(next_fname,bdos_func)) )
        {
        return 0;        /* Indicate no match */
        }
        /* Move file name and type */
movmem(pfname,&next_fname -> fcb_fname,11);
return 1;                /* Indicate match found */

} /*  End of get_nfn  */


/*===========================================================*/
char *srch_file(fcb,bdos_code)  /* Search for file */
/*===========================================================*/
/* This function issues either a search first or search next
   BDOS call. */

/* Entry Parameters */
struct _fcb *fcb;        /* pointer to file control block */          j
short bdos_code;         /* either SEARCHF or SEARCHN */

/* Exit parameters
   0 = no match found
   NZ = pointer to entry matched (currently in buffer)
*/
```

Figure 11-1. (Continued)

```
{
unsigned r_code;        /* Return code from search function
                           This is either 255 for no match, or 0, 1, 2, or 3
                           being the ordinal of the 32-byte entry in the
                           buffer that matched the name  */
char *dir_entry;        /* Pointer to directory entry */

        /* The BDS C compiler always sets the BDOS DMA
           to location 0x80 */

r_code = bdos(bdos_code,fcb);   /* Issue the BDOS call */
if (r_code == 255)              /* No match found */
        return 0;                                                  j

        /* Set a pointer to the matching
           entry by multiplying return code by 128
           and adding onto the buffer address (0x80),
           also add 1 to point to first character of name */

return (r_code << 5) + 0x81;

}/* End of srch_file */


/*==========================================================*/
rd_disk(drb)            /* Read disk (via BIOS) */
/*==========================================================*/
/* This function uses the parameters previously set up in the
   incoming request block, and, using the BIOS directly,
   executes the disk read. */

/* Entry parameters */
struct _drb *drb;       /* Disk request block (disk, track, sector, buffer) */

/* Exit parameters
   0 = No data available
   1 = Data available                                              k
*/

{
if (!set_disk(drb))     /* Call SELDSK, SETTRK, SETSEC */
        return 0;       /* If SELDSK fails, indicate
                           no data available */
if (bios(DREAD))        /* Execute BIOS read */
        return 0;       /* Indicate no data available if error returned */

return 1;               /* Indicate data available */

} /*  End of rd_disk  */


/*==========================================================*/
wrt_disk(drb)           /* Write disk (via BIOS) */
/*==========================================================*/
/* This function uses the parameters previously set up in the
   incoming request block, and, using the BIOS directly,
   executes the disk write. */

/* Entry parameters */
struct _drb *drb;       /* Disk request block (disk, track, sector, buffer) */

/* Exit parameters
   0 = Error during write
   1 = Data written OK                                             l
*/

{
if (!set_disk(drb))     /* Call SELDSK, SETTRK, SETSEC, SETDMA */
        return 0;       /* If SELDSK fails, indicate no data written */
if (bios(DWRITE))       /* Execute BIOS write */
        return 0;       /* Indicate error returned */

return 1;               /* Indicate data written */

} /*  End of wrt_disk  */
```

Figure 11-1. (Continued)

```
/*=============================================================*/
short set_disk(drb)     /* Set disk parameters */
/*=============================================================*/
/* This function sets up the BIOS variables in anticipation of
   a subsequent disk read or write. */

/* Entry parameters */
struct _drb *drb;       /* Disk request block (disk, track, sector, buffer) */

/* Exit parameters
   0 = Invalid disk (do not perform read/write)
   1 = BIOS now set up for read/write
*/

{
        /* The sector in the disk request block contains a
           LOGICAL sector. If necessary (as determined by the
           value in the disk parameter header), this must be
           converted into the PHYSICAL sector.
           NOTE: skewtab is declared as a pointer to a pointer to
           a short integer (single byte). */
short **skewtab;        /* Skewtab -> disk parameter header -> skew table */
short phy_sec;          /* Physical sector */
                                                                          m
        /* Call the SELDSK BIOS entry point. If this returns
           a 0, then the disk is invalid. Otherwise, it returns
           a pointer to the pointer to the skew table */
if ( !(skewtab = biosh(SELDSK,drb -> dr_disk)) )
        return 0;               /*  Invalid disk  */

bios(SETTRK,drb -> dr_track);   /* Set track */

        /* Note that the biosh function puts the sector into
           registers BC, and a pointer to the skew table in
           registers HL. It returns the value in HL on exit
           from the BIOS */
phy_sec = biosh(SECTRN,drb -> dr_sector,*skewtab); /* Get physical sector */
bios(SETSEC,phy_sec);           /* Set sector */
bios(SETDMA,drb -> dr_buffer);  /* Set buffer address */

return 1;               /* Indicate no problems */

} /* End of setp_disk */


/*      Directory Management Functions          */

/*=============================================================*/
get_nde(dir_pb)         /* Get next directory entry */
/*=============================================================*/
/* This function returns a pointer to the next directory entry.
   If the directory has not been opened, it opens it.
   When necessary, the next directory sector is read in.
   If the current sector has been modified and needs to be written back
   onto the disk, this will be done before reading in the next sector. */

/* Entry parameters */
struct _dirpb *dir_pb;          /* Pointer to the disk parameter block */

/* Exit Parameters
   Returns a pointer to the next directory entry in the buffer.          n
   The directory open and write sector flags in the parameter
   block are reset as necessary.
*/

{
if(!dir_pb -> dp_open)          /* Directory not yet opened */
        {
        if (!open_dir(dir_pb))  /* Initialize and open directory */
                {
                err_dir(O_DIR,dir_pb);          /* Report error on open */
                exit();
                }
                /* Deliberately set the directory entry pointer to the end
                   of the buffer to force a read of a directory sector */
```

Figure 11-1. (Continued)

```
        dir_pb -> dp_entry = dir_pb -> dp_buffer + DIR_BSZ;
        dir_pb -> dp_write = 0;         /* Reset write-sector flag */
        }

        /* Update the directory entry pointer to the next entry in
           the buffer. Check if the pointer is now "off the end"
           of the buffer and another sector needs to be read. */
if (++dir_pb -> dp_entry < dir_pb -> dp_buffer + DIR_BSZ)
        {
        return dir_pb -> dp_entry;      /* Return pointer to next entry */
        }

        /* Need to move to next sector and read it in */

        /* Do not check if at end of directory or move to
           the next sector if the directory has just been
           opened (but the opened flag has not yet been set) */
if (!dir_pb -> dp_open)
        dir_pb -> dp_open = 1;  /* Indicate that the directory is now open */
else
        {
        /* Check if the sector currently in the buffer needs to be
           written back out to the disk (having been changed) */
        if (dir_pb -> dp_write)
                {
                dir_pb -> dp_write = 0;         /* Reset the flag */
                if(!rw_dir(W_DIR,dir_pb))       /* Write the directory sector */
                        {
                        err_dir(W_DIR,dir_pb);  /* Report error on writing */
                        exit();
                        }
                }                                                            n

                /* Count down on number of directory entries left to process,
                   always four 32-byte entries per 128-byte sector */
        dir_pb -> dp_entrem -= 4;

                /* Set directory-end flag true if number of entries now < 0 */
        if (dir_pb -> dp_entrem == 0)           /* now at end of directory */
                {
                dir_pb -> dp_end = 1;           /* Indicate end */
                dir_pb -> dp_open = 0;          /* Indicate directory now closed */
                return 0;                       /* Indicate no more entries */
                }

                /* Update sector (and if need be track and sector) */
        if (++dir_pb -> dp_sector == dir_pb -> dp_sptrk)
                {
                ++dir_pb -> dp_track;           /* Update track */
                dir_pb -> dp_sector = 0;        /* Reset sector */
                }
        }

if(!rw_dir(R_DIR,dir_pb))       /* Read next directory sector */
        {
        err_dir(R_DIR,dir_pb);  /* Report error on reading */
        exit();
        }

        /* Reset directory-entry pointer to first entry in buffer */
return dir_pb -> dp_entry = dir_pb -> dp_buffer;

} /* End of get_nde */


/*===========================================================*/
open_dir(dir_pb)        /* Open directory */
/*===========================================================*/
/* This function "opens" up the file directory
   on a specified disk for subsequent processing
   by rw_dir, next_dir functions. */                                        o

/* Entry parameters */
struct _dirpb *dir_pb;  /* Pointer to directory parameter block */
```

Figure 11-1. (Continued)

```
/* Exit parameters
   0 = Error, directory not opened
   1 = Directory open for processing
*/

{
struct _dpb *dpb;                /* CP/M disk parameter block */

        /* Get disk parameter block address for the disk specified in
           the directory parameter block */
if ((dpb = get_dpb(dir_pb -> dp_disk)) == 0)
        return 0;       /* Return indicating no DPB for this disk */

        /* Set the remaining fields in the parameter block */
dir_pb -> dp_sptrk = dpb -> dpb_sptrk;  /* Sectors per track  */
dir_pb -> dp_track = dpb -> dpb_trkoff; /* Track offset of the directory  */
dir_pb -> dp_sector = 0;                /* Beginning of directory  */
dir_pb -> dp_nument = dpb -> dpb_maxden+1; /*  No. of directory entries  */
dir_pb -> dp_entrem = dir_pb -> dp_nument; /*  Entries remaining to process */
dir_pb -> dp_end = 0;                   /* Indicate not at end  */

        /* Set number of allocation blocks per directory entry to
           8 or 16 depending on the number of allocation blocks */
dir_pb -> dp_nabpde = (dpb -> dpb_maxabn > 255 ? 8 : 16);
        /* Set number of allocation blocks (one more than number of
           highest block) */
dir_pb -> dp_nab = dpb -> dpb_maxabn;

        /* Set the allocation block size based on the block shift.
           The possible values are: 3 = 1k, 4 = 2K, 5 = 4K, 6 = 8K, 7 = 16K.
           So a value of 16 is shifted right by (7 - bshift) bits. */
dir_pb -> dp_absize = 16 >> (7 - dpb -> dpb_bshift);

return 1;              /* Indicate that directory now opened */

} /*  End of open_dir  */
```

o

```
/*==============================================================*/
rw_dir(read_op,dir_pb)  /* Read/write directory */
/*==============================================================*/
/* This function reads/writes the next 128-byte
   sector from/to the currently open directory. */

/* Entry parameters */
short read_op;          /* True to read, false (0) to write */
struct _dirpb *dir_pb;  /* Directory parameter block */

/* Exit parameters
   0 = error -- operation not performed
   1 = operation completed
*/

{
struct _drb drb;                /* Disk request (for BIOS read/write) */


drb.dr_disk = dir_pb -> dp_disk;        /* Set up disk request */
drb.dr_track = dir_pb -> dp_track;
drb.dr_sector = dir_pb -> dp_sector;
drb.dr_buffer = dir_pb -> dp_buffer;

if (read_op)
        {
        if (!rd_disk(&drb))     /* Issue read command */
                return 0;       /* Indicate error -- no data available */
        }
else
        {
        if (!wrt_disk(&drb))    /* Issue write command */
                return 0;       /* Indicate error -- no data written */
        }
return 1;                       /* Indicate operation complete */

} /* End of rd_dir */
```

p

Figure 11-1. (Continued)

```
/*===========================================================*/
err_dir(opcode,dir_pb)          /* Display directory error
/*===========================================================*/
/* This function displays an error message to report an error
   detected in the directory management functions open_dir and rw_dir. */
/* Entry parameters */
short opcode;                   /* Operation being attempted */
struct _dirpb *dir_pb;  /* Pointer to directory parameter block */

{
printf("\n\007Error during ");

switch(opcode)
        {
        case R_DIR:
                printf("Reading");
                break;
        case W_DIR:
                printf("Writing");
                break;
        case O_DIR:
                printf("Opening");
                break;
        default:
                printf("Unknown Operation (%d) on",opcode);
        }

printf(" Directory on disk %c:. ",dir_pb -> dp_disk + 'A');

} /* End of err_dir */
```

q

```
/*===========================================================*/
setscb(scb,fname,user,extent,length)    /* Set search control block */
/*===========================================================*/
/* This function sets up a search control block according
   to the file name specified. The file name can take the
   following forms:

        filename
        filename.typ
        d:filename.typ
        *:filename.typ (meaning "all disks")
        ABCD...NOP:filename.typ (meaning "just the specified disks")

   The function sets the bit map according to which disks should be
   searched . For each selected disk, it checks to see if an error is
   generated when selecting the disk (i.e. if there are disk tables
   in the BIOS for the disk). */

/* Entry parameters */
struct _scb *scb;       /* Pointer to search control block */
char *fname;            /* Pointer to the file name */
short user;             /* User number to search for */
short extent;           /* Extent number to search for */
int length;             /* Number of bytes to compare */

/* Exit parameters
   None.
*/

{
int disk;               /* Disk number currently being checked */
unsigned adisks;        /* Bit map for active disks */

adisks = 0;             /* Assume no disks to search */

if (strscn(fname,":"))          /* Check if ":" in file name */
        {
        if (*fname == '*')      /* Check if "all disks" */
                {
                adisks = 0xFFFF;        /* Set all bits */
                }
        else                    /* Set specific disks */
                {
                while(*fname != ':')    /* Until ":" reached */
```

r

Figure 11-1. (Continued)

```
                        {
                        /* Build the bit map by getting the next disk
                           id. (A - P), converting it to a number in
                           the range 0 - 15, shifting a 1-bit left
                           that many places, and OR-ing it into the
                           current active disks. */
                        adisks |= 1 << (toupper(*fname) - 'A');
                        ++fname;         /* Move to next character */
                        }
                ++fname;                 /* Bypass colon */
                }
        }
else    /* Use only current default disk */
        {
                /* Set just the bit corresponding to the current disk */
        adisks = 1 << bdos(GETDISK);
        }

setfcb(scb,fname);      /* Set search control block as though it
                           were a file control block. */

/* Make calls to the BIOS SELDSK routine to make sure that
   all of the active disk drives have disk tables for them
   in the BIOS. If they don't, turn off the corresponding
   bits in the bit map. */

for (disk = 0;          /* Start with disk A: */
     disk < 16;         /* Until disk P: */
     disk++)            /* Use next disk */
        {
        if ( !((1 << disk) & adisks))
                continue;                /* Avoid selecting unspecified disks */
        if (biosh(SELDSK,disk) == 0)     /* Make BIOS SELDSK call */
                {                        /* Returns 0 if invalid disk */
                /* Turn OFF corresponding bit in mask
                   by AND-ing it with bit mask having
                   all the other bits set = 1  */
                adisks &= ((1 << disk) ^ 0xFFFF);
                }
        }

scb -> scb_adisks = adisks;     /* Set bit map in SCB */
scb -> scb_userno = user;       /* Set user number */
scb -> scb_extent = extent;     /* Set extent number */
scb -> scb_length = length;     /* Set number of bytes to compare */

} /* End setscb */
```
r

```
/*=============================================================*/
dm_clr(disk_map)                /* Disk map clear (to zeros) */
/*=============================================================*/
/* This function clears all elements of the disk map to zero. */

/* Entry Parameters */
unsigned disk_map[16][18];      /* Address of array of unsigned integers */

/* Exit parameters
   None.
*/

{
        /* WARNING -- The 576 in the setmem call below is based on
           the disk map array being [16][18] -- i.e. 288 unsigned
           integers, hence 576 bytes. */
setmem(disk_map,576,'\0');      /* Fill array with zeros */

} /* End of dm_clr */
```
s

```
/*=============================================================*/
dm_disp(disk_map,adisks)                /* Disk map display */
/*=============================================================*/
/* This function displays the elements of the disk map, showing
   the count in each element. A zero value-element is shown as
   blanks. For example:
```
t

Figure 11-1. (Continued)

```
     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  Used Free
A: 123      20  98          202     199 101 211                      954   70

   Lines will only be printed for active disks (as indicated by
   the bit map). */

/* Entry parameters */
unsigned disk_map[16][18];      /* Pointer to disk map array */
unsigned adisks;                /* Bit map of active disks */

{
#define USED_COUNT 16           /* "User" number for used entities */
#define FREE_COUNT 17           /* "User" number for free entities */

int disk;                       /* Current disk number */
int userno;                     /* Current user number */
unsigned dsum;                  /* Sum of entries for given disk */

printf("\n      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  Used Free");

for (disk = 0;                  /* Start with disk A: */
     disk < 16;                 /* Until disk P: */
     disk++)                    /* Next disk */
        {
        if (!(adisks & (1 << disk)))    /* Check if disk is active */
                continue;       /* No -- so bypass this one */

        printf("\n%c: ",disk + 'A');    /* Display disk number */

        dsum = 0;               /* Reset sum for this disk */
        for (userno = 0;        /* Start with user 0 */
             userno < 16;       /* Until user 15 */
             userno++)          /* Next user number */
                {
                dsum += disk_map[disk][userno]; /* Build sum */
                }

        if (dsum)       /* Check if any output for this disk,
                           and if not, display d: None */
                {
                /* Print either number or blanks */
                for (userno = 0;        /* Start with user 0 */
                     userno < 16;       /* Until user 15 */
                     userno++)          /* Next user number */
                        {
                        if (disk_map[disk][userno])
                                printf("%4d",disk_map[disk][userno]);
                        else
                                printf("    ");
                        }
                }
        else            /* No output for this disk */
                {
                printf(  " -- None --
                }
        printf("  %4d %4d",disk_map[disk][USED_COUNT],disk_map[disk][FREE_COUNT]);
        }

} /* End dm_disp */
```

t

```
/*==============================================================*/
get_dpb(disk)           /* Get disk parameter block address */
/*==============================================================*/
/* This function returns the address of the disk parameter
   block (located in the BIOS). */

/* Entry parameters */
char disk;              /* Logical disk for which DPB address is needed */

/* Exit parameters
        0 = Invalid logical disk
        NZ = Pointer to disk parameter block
*/

{
if (biosh(SELDSK,disk) == 0)            /* Make BIOS SELDSK call */
        return 0;                       /* Invalid disk */
```

u

Figure 11-1. (Continued)

```
    bdos(SETDISK,disk);                       /* Use BDOS SETDISK function */       u
    return bdos(GETDPARM);                    /* Get the disk parameter block */

    } /*  End of get_dpb  */


    /*      Code table functions    */                                            v

    /* Most programs that interact with a user must
       accept parameters from the user by name and translate
       the name into some internal code value.
       They also must be able to work in reverse, examining
       the setting of a variable, and determing what (ASCII
       name) it has been set to.

       An example is setting baud rates. The user may want to
       enter "19200," and have this translated into a number
       to be output to a chip. Alternatively, a previously
       set baud rate variable may have to be examined and the
       string "19200" generated to display its current
       setting to the user.

       A code table is used to make this task easier.
       Each element in the table logically consists of:
            A code value (unsigned integer)
            An ASCII character string (actually a pointer to it) */


    /*===========================================================*/
    ct_init(entry,code,string)      /* Initialize code table */
    /*===========================================================*/
    /* This function initializes a specific entry in a code table
       with a code value and string pointer.

"); NOTE: By convention, the last entry in a given
       code table will have a code value of CT_SNF (string not found). */

    /* Entry parameters */
    struct _ct *entry;                  /* Pointer to code table entry */
    int code;                           /* Code value to store in entry */
    char *string;                       /* Pointer to string for entry */

    /* Exit parameters
       None.
    */

    {
    entry -> _ct_code = code;                 /* Set _ct_code */
    entry -> _ct_sp = string;                 /* Set string pointer */
    } /*  end of ct_inti  */

    /*===========================================================*/             w
    unsigned
    ct_parc(table,string)           /* Parameter - return code */
    /*===========================================================*/
    /* This function searches the specified table for a
       matching string, and returns the code value that corresponds to it.
       If only one match is found in the table, then this function returns
       that code value. If no match or more than one match is found,
       it returns the error value, CT_SNF (string not found).
       This function is specifically designed for processing
       parameters on a command tail.
       Note that the comparison is done after conversion to uppercase
       (i.e. "STRING" matches "string"). A substring compare is used so
       that only the minimum number of characters for an unambiguous
       response need be entered. For example, if the table contained:

                    Code    Value
                    1       "APPLES"
                    2       "ORANGES"
                    3       "APRICOTS"

       A response of "O" would return code = 2, but "A" or "AP" would
       be ambiguous. "APR" or "APP" would be required. */

    struct _ct *table;                  /* Pointer to table */
    char *string;                       /* Pointer to key string */
```

Figure 11-1. (Continued)

```
{
int mcode;                      /* Matched code to return */
int mcount;                     /* Count of number of matches found */

mcode = CT_SNF;                 /* Assume error */
mcount = 0;                     /* Reset match count */

while(table -> _ct_code != CT_SNF) /* Not at end of table */
        {
        /* Compare keyboard response to table entry using
           uppercase substring compare. */
        if (usstrcmp(table -> _ct_sp,string) == 0)
                {
                mcount++;       /* Update match count */
                mcode = table -> _ct_code;      /* Save code */
                }
        table++;                /* Move to next entry */
        }

if (mcount == 1)                /* Only one match found */
        return mcode;           /* Return matched code */
else                            /* Illegal or ambiguous */
        return CT_SNF;

} /* End ct_parc */
```

w

```
/*==============================================================*/
unsigned
ct_code(table,string)   /* Return code for string */
/*==============================================================*/
/* This function searches the specified table for the
   specified string. If a match occurs, it returns the
   corresponding code value. Otherwise it returns CT_SNF
   (string not found).
   Unlike ct_parc, this function compares every character in the
   key string, and will return the code on the first match found. */

/* Entry parameters */
struct _ct *table;      /* Pointer to table */
char *string;           /* Pointer to string */

/* Exit parameters
   Code value -- if string found
   CT_SNF -- if string not found
*/

{
while(table -> _ct_code != CT_SNF)      /* For all entries in table */
        {
        if (ustrcmp(table -> _ct_sp,string) == 0) /* Compare strings */
                return table -> _ct_code;       /* Return code */
        table++;                        /* Move to next entry */
        }
return CT_SNF;                          /* String not found */

} /* End ct_code */
```

x

```
/*==============================================================*/
ct_disps(table) /*   Displays all strings in specified table */
/*==============================================================*/
/* This function displays all of the strings in a given table.
   It is used to indicate valid responses for operator input. */

/* Entry parameters */
struct _ct *table;              /* Pointer to table */

/* Exit Parameters
        None.
*/

{
while(table -> _ct_code != CT_SNF)      /* Not end of table */
        {
        printf("\n\t\t%s",table -> _ct_sp);     /* Print string */
        table++;                        /* Move to next entry */
        }
```

y

Figure 11-1. (Continued)

```
putchar('\n');                          /* Add final return */            y

} /* End of ct_disps */


/*===========================================================*/
ct_index(table,string) /* Returns index for a given string */
/*===========================================================*/
/* This function searches the specified table, and returns
   the INDEX of the entry containing a matching string.
   All characters of the string are used for the comparison,
   after they have been made uppercase. */


/* Entry parameters */
struct _ct *table;              /* Pointer to table */
char *string;                   /* Pointer to string */

/* Exit parameters
   Index of entry matching string, or
   CT_SNF if string not found.
*/                                                                        z

{
int index;                      /* Current value of index */

index = 0;                      /* Initialize index */

while(table -> _ct_code != CT_SNF)      /* Not at end of table */

        {
        if (ustrcmp(table -> _ct_sp,string) == 0)
                return index;   /* Return index */
        table++;                /* Move to next table entry */
        index++;                /* Update index */
        }
return CT_SNF;          /* String not found */

}


/*===========================================================*/
char *ct_stri(table,index)      /* Get string according to index */
/*===========================================================*/
/* This function returns a pointer to the string in the
   table entry specified by the index. */

/* Entry parameters */
struct _ct *table;              /* Pointer to table */                    aa
int index;                      /* Index into table */

{
struct _ct *entry;              /* Entry pointer */
        entry = table[index];   /* Point to entry */
        return entry -> _ct_sp; /* Return pointer to string */

} /* End of ct_stri */


/*===========================================================*/
char *ct_strc(table,code)  /* Get string according to code value */
/*===========================================================*/
/* This function searches the specified table and returns a
   pointer to the character string in the entry with the
   matching code value or a pointer to a string of "unknown"
   if the code value is not found. */

/* Entry parameters */                                                    bb
struct _ct *table;              /* Pointer to table */
unsigned  code;                 /* Code value */

{
while(table -> _ct_code != CT_SNF)      /* Until end of table */
        {
        if (table -> _ct_code == code)  /* Check code matches */
                return table -> _ct_sp; /* Yes, return ptr. to str. */
        table++;                        /* No, move to next entry */
```

Figure 11-1. (Continued)

```
        }
return "Unknown";
}



/*      Bit vector functions    */                                           bb

/* These functions manipulate bit vectors. A bit vector is a group
   of adjacent bits, packed eight per byte. Each bit vector has the
   structure defined in the LIBRARY.H file.

   Bit vectors are used primarily to manipulate the operating
   system's allocation vectors and other values that can best
   be represented as a series of bits. */


/*==========================================================*/
bv_make(bv,bytes)       /* Make a bit vector and clear to zeros */
/*==========================================================*/
/* This function uses C's built-in memory allocation, alloc,
   to allocate the necessary amount of memory, and then
   sets the vector to zero-bits. */

/* Entry parameters */
struct _bv *bv;         /* Pointer to a bit vector */
unsigned bytes;         /* Number of bytes in bit vector */

/* Exit parameter
   NZ = vector created
   0 = insufficient memory to create vector                                  cc
*/

{
if(!(bv -> bv_bits = alloc(bytes)))     /* Request memory */
        return 0;                       /* Request failed */

bv -> bv_bytes = bytes;                 /* Set length */
bv -> bv_end = bv -> bv_bits + bytes;   /* Set pointer to end */

bv_fill(bv,0);                          /* Fill with 0's */
return 1;

} /* End bv_make */


/*==========================================================*/
bv_fill(bv,value)       /* Fill bit vector with value */
/*==========================================================*/
/* This function fills the specified bit vector with the
   specified value.
   This function exist only for consistency's sake and
   to isolate the main body of code from standard
   functions like setmem. */

/* Entry parameters */
struct _bv *bv;         /* Pointer to bit vector */                          dd
char value;             /* Value to fill vector with */

/* Exit parameters
   None.
*/

{
/*     address      length         value */
setmem(bv -> bv_bits,bv -> bv_bytes,value);
}

/*==========================================================*/
bv_set(bv,bitnum)               /* Set the specified bit number */
/*==========================================================*/
/* This function sets the specified bit number in the bit vector
   to one-bit. */                                                            ee

/* Entry parameters */
struct _bv *bv;                 /* Pointer to bit vector */
unsigned bitnum;                /* Bit number to be set */
```

Figure 11-1. (Continued)

```
/* Exit parameters
   None.
*/

{
unsigned byte_offset;          /* Byte offset into the bit vector */

if ((byte_offset = bitnum >> 3) > bv -> bv_bytes)
       return 0;        /* Bitnum is "off the end" of the vector */

/* Set the appropriate bit in the vector. The byte offset
   has already been calculated. The bit number in the byte
   is calculated by AND ing the bit number with 0x07.
   The specified bit is then OR ed into the vector */

bv -> bv_bits[byte_offset] != (1 << (bitnum & 0x7));

return 1;               /* Indicate completion */

/* End of bv_set  */

/*==========================================================*/
bv_test(bv,bitnum)             /* Test the specified bit number */
/*==========================================================*/
/* This function returns a value that reflects the current
   setting of the specified bit. */

/* Entry parameters */
struct _bv *bv;                /* Pointer to bit vector */
unsigned bitnum;               /* Bit number to be set */

/* Exit parameters
   None.
*/

{
unsigned byte_offset;          /* Byte offset into the bit vector */

if ((byte_offset = bitnum >> 3) > bv -> bv_bytes)
       return 0;        /* Bitnum is "off the end" of the vector */

/* Set the appropriate bit in the vector. The byte offset
   has already been calculated. The bit number in the byte
   is calculated by AND ing the bit number with 0x07.
   The specified bit is then OR ed into the vector */

return bv -> bv_bits[byte_offset] & (1 << (bitnum & 0x7));

} /* End of bv_tests */


/*==========================================================*/
bv_nz(bv)               /* Test bit vector nonzero */
/*==========================================================*/
/* This function tests each byte in the specified vector,
   and returns indicating whether any bits are set in
   the vector. */

/* Entry parameters */
struct _bv *bv;         /* Pointer to bit vector */

/* Exit Parameters
   NZ = one or more bits are set in the vector
   0 = all bits are off
*/

{
char *bits;             /* Pointer to bits in bit vector */

bits = bv -> bv_bits;            /* Set working pointer */

while (bits != bv -> bv_end)     /* For entire bit vector */
        {
        if (*bits++)             /* If nonzero */
               return bits--;    /* Return pointer to NZ byte */
```

ee

ff

gg

Figure 11-1. (Continued)

```
        }
return 0;                       /* Indicate vector is zero */

} /* End of by_nz */
```
gg

```
/*============================================================*/
bv_and(bv3,bv1,bv2)             /* bv3 = bv1 & bv2 */
/*============================================================*/
/* This function performs a boolean AND between the bytes
   of bit vector 1 and 2, storing the result in bit vector 3. */

/* Entry parameters */
struct _bv *bv1;                /* Pointer to input bit vector */
struct _bv *bv2;                /* Pointer to input bit vector */

/* Exit parameters */
struct _bv *bv3;                /* Pointer to output bit vector */

{
char *bits1, *bits2, *bits3;    /* Working pointers to bit vectors */

bits1 = bv1 -> bv_bits;         /* Initialize working pointers */
bits2 = bv2 -> bv_bits;
bits3 = bv3 -> bv_bits;

        /* AND ing will proceed until the end of any one of the bit
           vectors is reached  */
while (bits1 != bv1 -> bv_end &&
       bits2 != bv2 -> bv_end &&
       bits3 != bv3 -> bv_end)
        {
                *bits3++ = *bits1++ & *bits2++; /* bv3 = bv1 & bv2 */
        }
} /* End of bv_and  */
```
hh

```
/*============================================================*/
bv_or(bv3,bv1,bv2)              /* bv3 = bv1 or bv2 */
/*============================================================*/
/* This function performs a boolean inclusive OR between the bytes
   of bit vectors 1 and 2, storing the result in bit vector 3. */

/* Entry parameters */
struct _bv *bv1;                /* Pointer to input bit vector */
struct _bv *bv2;                /* Pointer to input bit vector */

/* Exit parameters */
struct _bv *bv3;                /* Pointer to output bit vector */

{
char *bits1, *bits2, *bits3;    /* Working pointers to bit vectors */

bits1 = bv1 -> bv_bits;         /* Initialize working pointers */
bits2 = bv2 -> bv_bits;
bits3 = bv3 -> bv_bits;

        /* The OR ing will proceed until the end of any one of the bit
           vectors is reached. */
while (bits1 != bv1 -> bv_end &&
       bits2 != bv2 -> bv_end &&
       bits3 != bv3 -> bv_end)
        {
                *bits3++ = *bits1++ | *bits2++; /* bv3 = bv1 or bv2 */
        }
} /* End of bv_or */
```
ii

```
/*============================================================*/
bv_disp(title,bv)               /* Bit vector display */
/*============================================================*/
/* This function displays the contents of the specified bit vector
   in hexadecimal. It is normally only used for debugging. */

/* Entry parameters */
char *title;                    /* Title for the display */
struct _bv *bv;                 /* Pointer to the bit vector */
```
jj

Figure 11-1. (Continued)

```
/* Exit parameters
   None.
*/

{
char *bits;                     /* Working pointer */
unsigned byte_count;            /* Count used for formatting display */
unsigned bit_count;             /* Count for processing bits in a byte */
char byte_value;                        /* Value to be displayed */

printf("\nBit Vector : %s",title);      /* Display title */

bits = bv -> bv_bits;                   /* Set working pointer */
byte_count = 0;                         /* Initialize count */

while (bits != bv -> bv_end)            /* For the entire vector */
        {
        if (byte_count % 5 == 0)        /* Check if new line */
                                        /* Display bit number */
                printf("\n%4d : ",byte_count << 3);

        byte_value = *bits++;   /* Get the next byte from the vector */

        for (bit_count = 0; bit_count < 8; bit_count++)
                {
                /* Display the leftmost bit, then shift the value
                   left one bit */
                if (bit_count == 4) putchar(' '); /* Separator */
                putchar((byte_value & 0x80) ? '1' : '0');
                byte_value <<= 1;       /* Shift value left */
                }
        printf("  ");                           /* Separator */

        byte_count++;   /* Update byte count */
        }
} /* End of bv_disp */

/* End of LIBRARY.C */
```

Figure 11-1. (Continued)

Associated with the library of functions is another section of source code called "LIBRARY.H", shown in Figure 11-2. This "header" file must be included at the beginning of each program that calls any of the library functions.

For reasons of clarity, this chapter describes the simplest functions first, followed by the more complex, and finally by the utility programs that use the functions.

Several functions in the library and some definitions in the library header are not used by the utilities shown in this chapter. They have been included to illustrate techniques and because they might be useful in other utilities you could write.

```
#define LIBVN "1.0"     /* Library version number */

/* This file contains groups of useful definitions.
   It should be included at the beginning of any program
   that uses the functions in LIBRARY.C */


/* Definition to make minor language modification to C. */
#define short char              /* Short is not supported directly */
```

Figure 11-2. LIBRARY.H, code to be included at the beginning of any program that calls LIBRARY functions in Figure 11-1

```
/* One of the functions (bv_make) in the library uses the BDS C
   function, alloc, to allocate memory. The following definitions
   are provided for alloc. */

struct _header                  /* Header for block of memory allocated */
       {                                                                      b
       struct _header *_ptr;    /* Pointer to the next header in the chain */
       unsigned _size;          /* Number of bytes in the allocated block */
       };
struct _header _base;           /* Declare the first header of the chain */
struct _header *_allocp;        /* Used by alloc() and free() functions */


/* BDOS function call numbers  */

#define SETDISK 14      /* Set (select) disk */
#define SEARCHF 17      /* Search first */
#define SEARCHN 18      /* Search next */
#define DELETEF 19      /* Delete file */                                     c
#define GETDISK 25      /* Get default disk (currently logged in) */
#define SETDMA  26      /* Set DMA (Read/Write) Address */
#define GETDPARM 31     /* Get disk parameter block address */
#define GETUSER 32      /* Get current user number */
#define SETUSER 32      /* Set current user number */


/* Direct BIOS calls
   These definitions are for direct calls to the BIOS.
   WARNING:  Using these makes program less transportable.
   Each symbol is related to its corresponding jump in the
   BIOS jump vector.
   Only the more useful entries are defined. */

#define CONST   2       /* Console status */
#define CONIN   3       /* Console input */
#define CONOUT  4       /* Console output */
#define LIST    5       /* List output */
#define AUXOUT  6       /* Auxiliary output */
#define AUXIN   7       /* Auxiliary input */

#define HOME    8       /* Home disk */                                       d
#define SELDSK  9       /* Select logical disk */
#define SETTRK  10      /* Set track */
#define SETSEC  11      /* Set sector */
#define SETDMA  12      /* Set DMA address */
#define DREAD   13      /* Disk read */
#define DWRITE  14      /* Disk write */
#define LISTST  15      /* List status */
#define SECTRN  16      /* Sector translate */
#define AUXIST  17      /* Auxiliary input status */
#define AUXOST  18      /* Auxiliary output status */

                        /* "Private" entries in jump vector */
#define CIOINIT 19      /* Specific character I/O initialization */
#define SETDOG  20      /* Set watchdog timer */
#define CBGADDR 21      /* Configuration block, get address */

/* Definitions for accessing the configuration block */

#define CB_GET 21               /* BIOS jump number to access routine */
#define DEV_INIT 19             /* BIOS jump to initialize device */

#define CB_DATE 0               /* Date in ASCII */
#define CB_TIMEA 1              /* Time in ASCII */
#define CB_DTFLAGS 2            /* Date, time flags */
#define TIME_SET 0x01           /* This bit NZ means date has been set */     e
#define DATE_SET 0x02           /* This bit NZ means time has been set */

#define CB_FIP 3                /* Forced input pointer */
#define CB_SUM 4                /* System start-up message */

#define CB_CI 5                 /* Console input */
#define CB_CO 6                 /* Console output */
#define CB_AI 7                 /* Auxiliary input */
#define CB_AO 8                 /* Auxiliary output */
```

Figure 11-2. (Continued)

```
#define CB_LI 9                    /* List input */
#define CB_LO 10                   /* List output */

#define CB_DTA 11                  /* Device table addresses */
#define CB_C1224 12                /* Clock 12/24 format flag */
#define CB_RTCTR 13                /* Real time clock tick rate (per second) */

#define CB_WDC 14                  /* Watchdog count */
#define CB_WDA 15                  /* Watchdog address */

#define CB_FKT 16                  /* Function key table */
#define CB_COET 17                 /* Console output escape table */

#define CB_D0_IS 18                /* Device 0 initialization stream */
#define CB_D0_BRC 19               /* Device 0 baud rate constant */               e

#define CB_D1_IS 20                /* Device 1 initialization stream */
#define CB_D1_BRC 21               /* Device 1 baud rate constant */

#define CB_D2_IS 22                /* Device 2 initialization stream */
#define CB_D2_BRC 23               /* Device 2 baud rate constant */

#define CB_IV 24                   /* Interrupt vector */
#define CB_LTCBO 25                /* Long term config. block offset */
#define CB_LTCBL 26                /* Long term config. block length */

#define CB_PUBF 27                 /* Public files flag */
#define CB_MCBUF 28                /* Multi-command buffer */
#define CB_POLLC 29                /* Polled console flag */


        /* Device numbers and names for physical devices */
        /* NOTE: Change these definitions for your computer system */

#define T_DEVN 0                   /* Terminal */                                  f
#define M_DEVN 1                   /* Modem */
#define P_DEVN 2                   /* Printer */

#define MAXPDEV 2                  /* Maximum physical device number */

        /* Names for the physical devices */
                                                                                   g
#define PN_T "TERMINAL"
#define PN_M "MODEM"
#define PN_P "PRINTER"

        /* Structure and definitions for function keys */

#define FK_ILENGTH 2               /* No. of chars. input when func. key pressed
                                      NOTE: This does NOT include the ESCAPE. */
#define FK_LENGTH 16               /* Length of string  (not including fk_term) */
#define FK_ENTRIES 18              /* Number of function key entries in table */  h

struct _fkt                        /* Function key table */
        {
        char fk_input[FK_ILENGTH];       /* Lead-in character is not in table */
        char fk_output[FK_LENGTH];       /* Output character string */
        char fk_term;                    /* Safety terminating character */
        };


/* Definitions and structure for device tables */

        /* Protocol bits */
        /* Note: if the most significant bit is
           set = 1, then the set_proto function
           will logically OR in the value. This
           permits Input DTR to co-exist with                                      i
           XON or ETX protocol. */

#define DT_ODTR 0x8004             /* Output DTR high to send (OR ed in) */
#define DT_OXON 0x0008             /* Output XON */
#define DT_OETX 0x0010             /* Output ETX/ACK */

#define DT_IRTS 0x8040             /* Input RTS (OR-ed in) */
#define DT_IXON 0x0080             /* Input XON */
```

Figure 11-2. (Continued)

```
#define ALLPROTO 0xDC           /* All protocols combined */

struct _dt                      /* Device table */
        {
        char dt_f1[14];         /* Filler */
        char dt_st1;            /* Status byte 1 -- has protocol flags */
        char dt_st2;            /* Status byte 2 */
        unsigned dt_f2;         /* Filler */
        unsigned dt_etxml;      /* ETX/ACK message length */
        char dt_f3[12];         /* Filler */
        } ;                                                                      i


/* Values returned by the comp_fname (compare file name) */

#define NAME_EQ 0       /* Names equal */
#define NAME_LT 1       /* Name less than mask */                                j
#define NAME_GT 2       /* Name greater than mask */
#define NAME_NE 3       /* Name not equal (and comparison ambiguous) */


/* Structure for standard CP/M file control block */

#define FCBSIZE 36              /* Define the overall length of an FCB */

struct _fcb
        {
        short fcb_disk;         /* Logical disk (0 = default) */
        char fcb_fname[11];     /* File name, type (with attributes) */
        short fcb_extent;       /* Current extent */
        unsigned fcb_s12;       /* Reserved for CP/M */                          k
        short fcb_reccnt;       /* Record count used in current extent */
        union                   /* Allocation blocks can be either */
                {               /* Single or double bytes */
                short fcbab_short[16];
                unsigned fcbab_long[8];
                } _fcbab;
        short fcb_currec;       /* Current record within extent */
        char fcb_ranrec[3];     /* Record for random read/write */
        };

/*  Parameter block used for calls to the directory management routines  */

#define DIR_BSZ 128             /* Directory buffer size */

struct _dirpb
        {
        short dp_open;          /* 0 to request directory to be opened */
        short dp_end;           /* NZ when at end of directory */
        short dp_write;         /* NZ to write current sector to disk */
        struct _dir *dp_entry;  /* Pointer to directory entry in buffer */      l
        char dp_buffer [DIR_BSZ];       /* Directory sector buffer */
        char dp_disk;           /* Current logical disk */
        int dp_track;           /* Start track */
        int dp_sector;          /* Start sector */
        int dp_nument;          /* Number of directory entries */
        int dp_entrem;          /* Entries remaining to process */
        int dp_sptrk;           /* Number of sectors per track */
        int dp_nabpde;          /* Number of allocation blocks per dir. entry */
        unsigned dp_nab;        /* Number of allocation blocks */
        int dp_absize;          /* Allocation block size (in Kbytes) */
        };


/* The err_dir function is used to report errors found by the
   directory management routines, open_dir and rw_dir.
   Err_dir needs a parameter to define the operation being
   performed when the error occurred. The following definitions
   represent the operations possible. */                                         m

#define W_DIR   0       /* Writing directory */
#define R_DIR   1       /* Reading directory */
#define O_DIR   2       /* Opening directory */
```

Figure 11-2. (Continued)

```
/*  Disk parameter block maintained by CPM  */                                  n

struct _dpb
        {
        unsigned dpb_sptrk;      /* Sectors per track */
        short dpb_bshift;        /* Block shift */
        short dpb_bmask;         /* Block mask */
        short dpb_emask;         /* Extent mask */
        unsigned dpb_maxabn;     /* Maximum allocation block number */
        unsigned dpb_maxden;     /* Maximum directory entry number */
        short dpb_rab0;          /* Allocation blocks reserved for */
        short dpb_rab1;          /*   directory blocks */
        unsigned dpb_diskca;     /* Disk changed workarea */
        unsigned dpb_trkoff;     /* Track offset */
        };

/*  Disk directory entry format  */                                             o

struct _dir {
        char de_userno;          /* User number or 0xE5 if free entry */
        char de_fname[11];       /* File name [8] and type [3] */
        int de_extent;           /* Extent number of this entry */
        int de_reccnt;           /* Number of 128-byte records used in last
                                      allocation block */
        union                    /* Allocation blocks can be either */
                {                /*   single or double bytes */
                short de_short[16];
                unsigned de_long[8];
                } _dirab;
        };

/*  Disk request parameters for BIOS-level read/writes */                       p

struct _drb
        {
        short dr_disk;           /* Logical disk A = 0, B = 1... */
        unsigned dr_track;       /* Track (for SETTRK) */
        unsigned dr_sector;      /* Sector (for SETSEC) */
        char *dr_buffer;         /* Buffer address (for SETDMA) */
        } ;

/* Search control block used by directory scanning functions */                 q

struct _scb
        {
        short scb_userno;        /* User number(s) to match */
        char scb_fname[11];      /* File name and type */
        short scb_extent;        /* Extent number */
        char unused[19];         /* Dummy bytes to make this look like
                                      a file control block */
        short scb_length;        /* Number of bytes to compare */
        short scb_disk;          /* Current disk to be searched */
        unsigned scb_adisks;     /* Bit map of disks to be searched.
                                      the rightmost bit is for disk A:. */
        } ;

/* Code table related definitions */                                            r

#define CT_SNF 0xFFFF   /* String not found */

struct _ct              /* Define structure of code table */
        {
        unsigned _ct_code;       /* Code value */
        char *_ct_sp;            /* String pointer */
        };
```

Figure 11-2. (Continued)

```
/* Structure for bitvectors */

struct _bv
        {
        unsigned bv_bytes;          /* Number of bytes in the vector */
        char *bv_bits;              /* Pointer to the first byte in the vector */       s
        char *bv_end;               /* Pointer to byte following bit vector */
        } ;


/* End of LIBRARY.H */
```

Figure 11-2. (Continued)

Library Functions

This section describes the library functions and the sections from the header file that must be included at the beginning of each utility program.

A Minor Change to C Language

One minor problem with the BDS C Compiler is that it does not support "short" integers, or integers that are only a single byte long. It is convenient to declare certain values as short to serve as a reminder of the standard type definition. Therefore, the BDS C compiler must be "fooled" by declaring these values to be single characters. To do this, the library header file contains the declaration

```
#define short char.
```

shown in Figure 11-2, section a.

The "#define" tells the first part of the C compiler, the preprocessor, to substitute the string "char" (which declares a character variable) whenever it encounters the string "short" (which would ordinarily declare a short integer in standard C).

Note that character strings enclosed in "/*" and "*/" are regarded as comments and are ignored by the compiler.

BDOS Calls

The standard library of functions that comes with the BDS C compiler includes a function to make BDOS calls, called "bdos." It takes two parameters, and a typical call is of the following form:

```
bdos(c,de);
```

The "c" parameter represents the value that will be placed into the C register. This is the BDOS function code number. The "de" is the value that will be placed in the DE register pair.

The library header contains definitions (#define declarations) for BDOS functions 14 through 32, making these functions easier to use (Figure 11-2, c). Function 32 (Get/Set Current User Number) has two definitions; the "de" parameter is used to differentiate whether a get or a set function is to be performed.

BIOS Calls

The BDS C standard library also contains two functions that make direct BIOS calls. These are "bios" and "biosh." They differ only in that the bios function returns the value in the A register on return from the BIOS routine, whereas biosh, as its name implies, returns the value in the HL register pair. Examples of their use are

```
bios(jump_number,bc);
```

and

```
biosh(jump_number,bc,de);
```

Both functions take as their first parameter the number of the jump instruction in the BIOS jump vector to which control is to be transferred. For example, the console-status entry point is the third JMP in the vector. Numbering from 0, this would be jump number 2.

The library header file contains #defines for BIOS jumps 2 through 21 (Figure 11-2, d). The last group of these #defines (19 through 21) is for the "private" additions to the standard BIOS jump vectors described in Chapter 8.

Remember, though, that using direct BIOS calls makes programs more difficult to move from one system to another.

BIOS Configuration Block Access

As you may recall, the configuration block is a collection of data structures in the BIOS. These structures are used either to store the current settings of certain user-selectable options, or to point to other important data structures in the BIOS.

One of the "private" jumps appended to the standard BIOS jump vector transfers control to a routine that returns the address in memory of a specified data structure. For example, if a utility program needs to locate the word in the BIOS that determines from which physical device the console input is to read, it can transfer control to jump 21 in the BIOS jump vector (actually the 22nd jump) with a code value of 5 in the C register. This jump transfers control to the CBGet-Address code, which on its return will set HL to the address of the console input redirection vector. The utility program can then read from or write into this variable. The library header file contains #define declarations relating the code values to mnemonic names (Figure 11-2, e).

You will need to refer to the source code in Figure 8-10 to determine whether the address returned by the BIOS function is the address of the data element or the address of a higher-level table that in turn points to the data element.

In order to access the current system date, for example, you would include the following code:

```
char *ptr_to_date;               /* declare date pointer*/
ptr_to_date = biosh(CB_DATE);    /* get address */
```

The ptr_to_date can then be used to access the date directly.

During initial debugging of a utility, it is useful to be able to intercept all such accesses to the configuration block, partly to reassure yourself that the utility program is working as it should, and partly to ensure that the BIOS routine is returning the correct addresses to the data structures. Therefore, the utility library contains a function, "get_cba," that gets a configuration block address (Figure 11-1, a).

At first, it appears that get_cba is declared as a function that returns a pointer to characters. This is not strictly true. Sometimes the address it returns will point to characters, sometimes to integers, and sometimes to structures (such as the function key table).

The "printf" instruction has been left in the function in anticipation of debugging a utility. If you need to see some debug output whenever the get_cba function is used, delete the "/*" and "*/" surrounding the "printf" and recompile the library.

BIOS Function Key Table Access

The BIOS shown in Figure 8-10 contains code to recognize when an incoming escape sequence indicates that one of the terminal's function keys has been pressed. Instead of returning just the escape sequence, the console driver injects a previously programmed string of characters into the console input stream. For example, on a DEC VT-100 terminal, when the PF1 function key is pressed, the terminal emits the following character sequence: ESCAPE, "O", "P". The function key table contains the "OP" and a 00H-byte-terminated string of characters to be injected into the console input stream. In Figure 8-10, the example string is "FUNCTION KEY 1", LINE FEED. The library header file contains a declaration for the structure of the function key table (Figure 11-2, h).

Note the use of "#define" to declare the length of the incoming characters emitted by the terminal as well as the length of the output string.

In order to access a function key table entry, you must declare a pointer to a "_fkt" structure like this:

```
struct _fkt *ptr_to_fkt;      /* Declare pointer */
ptr_to_fkt = get_cba(CB_FKT); /* Set pointer */
printf("Display the first string : %s",
     ptr_to_fkt -> fk_output);
++ptr_to_fkt;                 /* Move to next entry */
```

The get_cba function is used to return the address of the first entry in the function key table and set a pointer to it. Then the printf function (part of the standard BDS C library) is used to print out the first string, which gets substituted for the "%s" in the quoted string. Note that the statement

```
++ptr_to_fkt
```

does not just add one to the pointer to the function key table—it adds whatever it takes to move the pointer to the next *entry* in the table.

BIOS Device Table Access

The device tables are important structures for the serial devices served by the console, auxiliary, and list device drivers in the BIOS. They are declared at line 1500 in Figure 8-10.

The get_cba function does not return a pointer to a specific device table, but a pointer to a table of device table addresses. Each entry in the address table corresponds to a specific device number. If there is no device table for a specific device number, then the corresponding entry in the table will be set to zero. the library header file contains definitions for the device table (Figure 11-2, i).

The device tables contain, among other things, the current serial line protocols used to synchronize the transmission and reception of data by the device drivers and the physical devices. An example utility, PROTOCOL, is shown later in the chapter. The example #define declarations and structure definition shown here are modeled on the requirements of this utility. The only relevant bytes are the two status bytes dt_st1 and dt_st2 and the message length used with the ETX/ACK protocol, dt_etxml. The #defines shown are for the specific bits in the device table's status bytes. The PROTOCOL utility uses the most significant bit to indicate whether a given protocol setting can coexist with others.

To access these fields, use the following code:

```
struct _ppdt
     {
     char *pdt[16];        /* Array of 16 pointers to device tables */
     } *ppdt;              /* Pointer to array of 16 pointers */
struct _dt *dt;            /* Pointer to device table */

ppdt = get_cba(CB_DTA);        /* Set pointer to array of pointers */
dt = ppdt -> pdt[device_no];   /* Set pointer to specified device
                                  table */

if (!dt)
     printf("\nError - no device table for this device.");

dt -> dt_etxml = 0;            /* Clear ETX message length */
```

BIOS Disk Parameter Block Access

Several of the utility programs shown in this chapter must access the file directory on a given logical disk. The disk parameter block (DPB) indicates the size and location of the file directory. The library header contains a structure definition that describes the DPB (Figure 11-2, n).

To locate the DPB, you can make a direct BIOS call to the SELDSK routine, which returns the address of the disk parameter header (DPH). You then can access the DPB pointer in the DPH. Alternatively, using the BDOS, you can make the required disk the default disk and then request the address of its DPB. The code for the latter method is shown in the get_dpb function included in the utility library (Figure 11-1, u).

The get_dpb function uses a BIOS SELDSK function first to see if the specified disk is legitimate. Only then does it use the BDOS.

Reading or Writing a Disk Using the BIOS

When you write a program that uses direct BIOS calls, you increase the possibility of problems in moving the program from one system to another. However, in certain circumstances it is necessary to use the BIOS. Reading and writing the file directory is one of these; the BDOS cannot be used to access the directory directly. The library header contains a structure declaration for a parameter block that contains the details of an "absolute" disk read or write (Figure 11-2, p).

Note the pointer to the 128-byte data buffer used to hold one of CP/M's "records."

The disk read and write functions are rd_disk (Figure 11-1, k) and wrt_disk (Figure 11-1, l). Both of them take a _drb as an input parameter, and both call the set_disk function to make the individual BIOS calls to SELDSK, SETTRK, and SETSEC.

Of special note is the code in set_disk (Figure 11-1, m) that converts a logical sector into a physical sector using the sector translation table and the SECTRAN entry point in the BIOS.

File Directory Entry Access

All of the utility programs that access a disk directory share the same basic logic regardless of their specific task. This logic can be described best in pseudocode:

```
while (not at the end of the directory)
   {
   access the next directory entry
   if (this entry matches the current search criteria)
      {
      process the entry
      }
   }
```

There are two ways of implementing this logic. The first uses the BIOS to read the directory. Entries are presented to the utility exactly as they occur in the file directory. The second uses the BDOS functions Search First and Search Next and accesses the directory file-by-file rather than by entry. This latter method is more suited to utilities that process files rather than entries. The ERASE utility, described later in this chapter, illustrates this second method.

Three groups of functions are provided in the library: to access the next entry in the directory, to match the name in the current entry against a search key, and to assist with processing the directory.

Directory Accessing Functions

A number of functions involve access to the file directory. The first group of such functions performs the following:

get_nde (get next directory entry; Figure 11-1, n)
This function returns a pointer to the next directory entry, or returns zero if the end of the directory has been reached.

open_dir (open directory; Figure 11-1, o)
This function is called by get_nde to open up a directory for processing.

rw_dir (read/write directory; Figure 11-1, p)
This function reads or writes the current directory sector.

err_dir (error on directory; Figure 11-1, q)
This general-purpose routine displays an error message if the BIOS indicates that it had problems either reading or writing the directory.

All of these functions use a directory parameter block to coordinate their activity. The library header contains the definitions for this structure (Figure 11-2, l), as well as #define declarations for operation codes used by the directory-accessing functions (Figure 11-2, m).

Before calling get_nde, the calling program needs to set dp_open to zero (forcing a call to open_dir) and the dp_disk field to the correct logical disk. The open_dir function sets up all of the remaining fields, using get_dpb to access the disk parameter block for the disk specified in dp_disk.

Of the remaining flags, dp_end will be set to true, when the end of the directory is reached, and dp_write must be nonzero for rw_dir to write the current sector back onto the disk.

The get_nde function includes all of the necessary logic to move from one directory entry to the next, reading in the next sector when necessary, and writing out the previous sector if the dp_write flag has been set to a nonzero value by the calling program. It also counts down on the number of directory entries processed, detecting and indicating the end of the directory.

The code at the beginning of the function calls open_dir if the dp_open flag is false. Note the code at the end of open_dir that sets the number of allocation blocks per directory entry (dp_nabpde). This number is computed from the maximum allocation block number in the disk parameter block. If it is larger than 255, each allocation block must occupy a word, and there will be eight blocks per directory entry. If there are 255 or fewer allocation blocks, each will be one byte long and there will be 16 per entry. The allocation block size, in Kbytes, is computed from a simple formula.

In the early stages of debugging utilities, comment out the line that makes the call to wrt_disk. This will prevent the directory from being overwritten. You then can test even those utilities that attempt to erase entries from the directory without any risk of damaging any data on the disk.

The last function in this group, err_dir, is a common error handling function for taking care of errors while reading or writing the directory.

Directory Matching Functions

The second group of functions that access the file directory matches each directory entry against specific search criteria. These include the following functions:

setscb (set search control block; Figure 11-1, r)
A search control block (SCB) is a structure that defines the entries in the directory that are to be selected for processing.

comp_fname (compare file name; Figure 11-1, f)
This function compares the file name in the current directory entry with the one specified in the search control block.

The library header contains the structure definition for the search control block (Figure 11-2, q). This SCB is a hybrid structure. The first part of it is a cross between a file control block (FCB) and a directory entry. The last three fields, scb_length, scb_disk, and scb_adisks, are peculiar to the search control block. Note that its overall length is the same as an FCB's so that the standard BDS C function set_fcb can be used. This function sets the file name and type into an FCB, replacing "*" with as many "?" characters as are required, and clears all unused bytes to zero.

The scb_length field indicates to the comp_fname (compare file name) function how many bytes of the structure are to be compared. This field will be set to 12 to compare the user number, file name, and type, or to 13 to include the extent number.

Note that scb_disk is the *current* disk to be searched, whereas scb_adisks is a bit map with a 1 bit corresponding to each of the 16 possible logical disks that must be searched.

The search control block is initialized by the setscb function.

Note the form of the file name that setscb expects to receive. This is described in the comments at the beginning of the function.

Several of the utility programs use their own special versions of setscb, renaming it ssetscb (special setscb) to avoid the library version being linked into the programs.

The complementary function comp_fname is used to compare the first few bytes of the current directory entry to the corresponding bytes of the SCB.

The comp_fname function performs a specialized string match of the user number, the file name, the file type, and, optionally, the extent number. A "?" character in the search control block file name, type, and extent will match with any character in the file directory entry. However, in the SCB user number, a "?" will only match a number in the range 0 to 15; it will not match a directory entry that has the user number byte set to E5H (or 0xE5, as hexadecimal notation in C).

This function also returns one of several values to indicate the result of the comparison. These values are defined in the library header file (Figure 11-2, j).

Directory Processing Functions

The final group of functions that access the directory are those that help process the directory entries themselves. These functions use a structure definition to access each directory entry (Figure 11-2, o).

A union statement is used for the allocation block numbers. These can be single- or two-byte entries, depending on the maximum number of allocation blocks that must be represented. The union statement tells the BDS C compiler whether there will be a 16-byte array of short integers (characters) or an array of eight unsigned two-byte integers.

The functions contained in this group can be divided into three subgroups:

- Those that deal with converting directory entries for display on the console.
- Those that deal with a "disk map"—a convenient array for representing logical disks and the user numbers they contain.
- Those that deal with "bit vectors"—a convenient representation of which allocation blocks on a logical disk are in use or available.

The library contains only one function to convert a directory-entry file name into a suitable form for display on the console. This is the conv_dfname function (Figure 11-1, h). It takes the information from the specified directory entry (or, as a convenience, a search control block) and formats it into a string of the form

```
uu/d:filename.typ
```

The "uu" specifies the user number and the "d" specifies the disk identification.

The repetitive code at the end of the function is necessary to make sure that the characters in the file type do not have their high-order bits set. These bits are the file attributes. If they are set, they can render the characters nondisplayable on some terminals.

The second subgroup of functions, those that manipulate a "disk map," produce an array that looks like this:

```
Disks
  :
  v  User Numbers -->                                         -Totals-
  A  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 Used Free
  B
  :
  :
  P
```

This disk map is used by several utility programs. For example, the SPACE utility displays a disk map that shows, for each logical disk in the system, and for each user on each logical disk, how many Kbytes of disk space are in use. The totals at the right show the total of used and free space. In another example, the FIND utility shows how many files on each disk and in each user number match the search name.

Each utility program that uses a disk map is coded:

```
unsigned disk_map[16][18];
```

Two functions are provided in the library to deal with the disk map:

dm_clr (disk map clear; Figure 11-1, s)
: This function fills the entire disk map with zeros.

dm_disp (disk map display; Figure 11-1, t)
: This function displays the horizontal and vertical caption lines for the disk map and then converts each element of the disk map to a decimal number.

The first function, dm_clr, uses one of the standard BDS C functions to set a block of memory to a specific value. It presumes that the disk map is 16 × 18 elements, each two bytes long.

The second function, dm_disp, prints horizontal lines only for those disks specified in the bit map parameter. Here is an example of its output:

```
       0    1    2    3    4   ...   10   11   12   13   14   15  Used Free
A:     1    1                                                      15  241
B:    66   20   74   50    3                                      245  779
C:    -- None --                                                    0 1024
(NOTE: All user groups would be shown on the terminal.)
```

The final subgroup deals with processing "bit vectors." A bit vector is a string of bits packed eight bits per byte. Each bit is addressed by its relative number along the vector; the first bit is number 0.

An example of why bit vectors are used is a utility program that needs to scan the directory of a disk and build a structure showing which allocation blocks are in use. It can do this by accessing each active directory element and, for each nonzero allocation block number, setting the corresponding bit number in a bit vector.

The library header has a structure definition for a bit vector (Figure 11-2, s).

This vector contains the overall length of the bit vector in bytes, and two pointers. The first points to the start of the vector, the second to the end. The bytes that contain the vector bits themselves are allocated by the alloc function — one of the standard BDS C functions.

The following bit vector functions are provided in the library:

bv_make (bit vector make; Figure 11-1, cc)
: This function allocates memory for the bit vector (using the standard mechanism provided by BDS C) and sets all of the bits to zero.

bv_fill (bit vector fill; Figure 11-1, dd)
: This fills a specified vector, setting each byte to a specified value.

bv_set (bit vector set; Figure 11-1, ee)
: This sets the specified bit of a vector to one.

bv_test (bit vector test; Figure 11-1, ff)
: This function returns a value of zero or one, reflecting the setting of the specified bit in a bit vector.

bv_nz (bit vector nonzero; Figure 11-1, gg)
: This returns zero or a nonzero value to reflect whether *any* bits are set in the specified bit vector.

bv_and (bit vector AND; Figure 11-1, hh)
: This function performs a Boolean AND between two bit vectors and places the result into a third vector.

bv_or (bit vector OR; Figure 11-1, ii)
: This is similar to bv_and, except that it performs an inclusive OR on the two input vectors.

bv_disp (bit vector display; Figure 11-1, jj)
: This function displays a caption line and then prints out the contents of the specified bit vector as a series of zeros and ones. Each byte is formatted to make the output easier to read.

The bv_make function uses the alloc function to allocate a block from the unused part of memory between the end of a program and the base of the BDOS. It requires that two data structures be declared at the beginning of the program. These structures are declared in the library header file (Figure 11-2, b).

The bv_fill function uses the standard BDS C setmem function.

The bv_set function converts the bit number into a byte offset by shifting the bit number right three places. The least significant three bits of the original bit number specify which bit in the appropriate byte needs to be ORed in.

The bv_test function is effectively the reverse of bv_set. It accesses the specified bit and returns its value to the calling program.

The bv_nz function scans the entire bit vector looking for the first nonzero byte. If the entire vector is zero, it returns a value of zero. Otherwise, it returns a pointer to the first nonzero byte.

Both bv_and and bv_or functions take three bit vectors as parameters. The first vector is used to hold the result of either ANDing or ORing the second and third vectors together. Both of these functions assume that the output vector has already been created using bv_make. The shortest of the three vectors will terminate the bv_and or bv_or function; that is, these functions will terminate when they reach the end of the first (shortest) vector.

The final function, bv_disp, displays the title line specified by the calling program, and then displays all of the bits in the vector, with the bit number of the first bit on each line shown on the left.

None of the utility programs uses bv_disp—it has been left in the library purely as an aid to debugging.

Here is an example of bv_disp's output:

```
Bit Vector : Allocation Blocks in Use
   0 : 0000 0000  0001 1000  1000 0001  1111 1111  1111 1111
  40 : 1111 1111  1111 1111  1111 1111  1110 1011  0000 0000
  80 : 1100 0000  1111 1100  1111 1001  1100 0000  1001 1111
 120 : 1110 1100  0001 1111  0000 0000  1101 1000  0001 1110
 160 : 1111 1111  1110 1111  1110 1111  0000 0111  0000 0111
 200 : 1111 0010
```

Checking User-Specified Parameters

The C language provides a mechanism for accessing the parameters specified in the "command tail." It provides a count of the number of parameters entered, "argc" (argument count), and an array of pointers to each of the character strings, "argv" (argument vector). At the beginning of the main function of each program you must define these two variables like this:

```
main(argc,argv)
{
int argc;       /* Argument count */
char *argv[];   /* Array of pointers to char. strings */
:
: /* Remainder of main function */
:
}
```

Consider the minimum case—a command line with just the program name on it:

```
A>command
```

The convention is that the first argument on the line is the name of the program itself. Hence argc would be set to one, and argv[0] would be a pointer to the program name, "command."

Next consider a more complex case—a command line with parameters like the following:

```
A>command param1 123
```

In this case, argc will be three; argv[1] will be a pointer to param1; and argv[1][0] will access the 0 (the first) character of argv[1]—in this case the character "p."

To detect whether the second parameter is present and numeric, the code will be

```
if (isdigit(argv[1][0]))
     {
          /* Process digit */
     }
else
     {
          /* Parameter either not present or has
             alpha character at the front */
     }
```

In most of the utilities, you will get a much "friendlier" program if the user need only specify enough characters of a parameter to distinguish the value entered from the other possible values. For example, consider a program that can have as a parameter one of the following values: 300, 600, 1200, 2400, 4800, 9600, or 19200. It would be convenient if the user needed to type only the first digit, rather than having to enter redundant keystrokes. However, the values 1200 and 19200 would then be ambiguous. The user would have to enter 12 or 19. Novice users often prefer to specify the entire parameter for clarity and security.

The standard C library provides a character string comparison function, strcmp. Unfortunately, this function does not provide for the partial matching just described. Therefore, the library includes two special functions that do make this possible: sstrcmp (substring compare, Figure 11-1, d) and usstrcmp (uppercase substring compare, Figure 11-1, e). The latter function is necessary when you need to compare a substring that could contain lowercase characters; it converts characters to uppercase before the comparison.

To assist with character string manipulation, two additional functions have been included in the library. These are strscn (string scan, Figure 11-1, b) and ustrcmp (uppercase string compare, Figure 11-1, c).

Using Code Tables

A code table is a simple structure used by all of the utility programs that accept parameters that can have any of several values. The library header contains a structure definition for a code table (Figure 11-2, r).

A code table entry contains an unsigned code value and a pointer to a character string. It is used in the utility programs wherever there is a need to relate some arbitrary code number or bit pattern to an ASCII character string. For example, to program a serial port baud-rate-generator chip to various baud rates requires different time constants for each rate. Users do not need to know what these numbers are; they only need to be able to specify the baud rate as an ASCII string.

Thus, a code table is set up as follows:

Baud Rate Constant	**User's Name**
0x35	"300"
0x36	"600"
0x37	"1200"
0x3A	"2400"
0x3C	"4800"
0x3E	"9600"
0x3F	"19200"

A utility program now needs to be able to perform various operations using the code table:

- Given the input parameter on the command tail, the utility must check whether the ASCII string is in the code table, display all of the legal options on the console if it is not, and return the code value for subsequent processing if it is.
- Given the current baud rate constant (held in the BIOS), the utility must scan the code table and display the corresponding ASCII string to tell the user the current baud rate setting.

The library includes specialized functions to do this, plus some additional functions to make code tables more generally usable. These functions are

ct_init (code table initialize; Figure 11-1, v)
This function initializes a specific entry in a code table, setting the code value and the pointer to the character string.

ct_parc (code table parameter return code; Figure 11-1, w)
This performs an uppercase substring match on the specified key string, returning either an error (the value CT_SNF—string not found) or a code value.

ct_code (code table return code; Figure 11-1, x)
This function is similar to ct_parc in that it scans a code table and returns the corresponding code. It differs in the way that the comparison is done. The entire search string is compared with the string in the code table entry. A match only occurs when all characters are the same.

ct_disps (code table display strings; Figure 11-1, y)
This function displays all strings in a given code table. It is used either when the user has entered an invalid string, or when the utility program is requested to show what options are available for a parameter.

ct_index (code table return index; Figure 11-1, z)
This function, given a string, searches the code table and returns the *index*

of the entry that has a string matching the search string. The index is not the code value; it is the number of the entry in the table.

ct_stri (code table string index; Figure 11-1, aa)
This function, given an entry index number, returns a pointer to the string in that entry.

ct_strc (code table string code; Figure 11-1, bb)
This function, given a code number, returns a pointer to the string in the entry that has a matching code number.

Accessing a Directory via the BDOS

One problem associated with accessing the file directory directly, as illustrated by earlier functions, is that the program is presented with directory entries in exactly the order that they occur in the directory. For some programs, such as those that process groups of files, it is better to use the BDOS Search First and Search Next functions to access the directory.

Using the BDOS, the program can process the first file name to match an ambiguous search key, then go back to the BDOS to get the name of the next file, and so on. The library header contains a structure definition for a standard CP/M file control block (Figure 11-2, k).

Notice that the first byte of the FCB is a disk number rather than the user number of the directory entry. Note also the use of a union statement to describe the allocation block numbers.

The standard BDS C library contains a function, setfcb, that is given the address of an FCB and a pointer to a string containing a file name. It converts any "*" in the name to the appropriate number of "?", and fills the remainder of the FCB with zeros.

The example library contains the following functions designed for BDOS file directory access:

get_nfn (get next file name; Figure 11-1, i)
This function is given a pointer to an ambiguous file name and a pointer to an FCB. It returns with the FCB set up to access the next file that matches the ambiguous file name.

srch_file (search for file; Figure 11-1, j)
This function, used by get_nfn, issues either a Search First or a Search Next BDOS call.

conv_fname (convert file name; Figure 11-1, g)
This function converts a file name from an FCB into a form suitable for display on the console. It is similar to the conv_dfname function described earlier except that it outputs only the disk, file name, and type (not the user number) in the form

```
d:filename.typ
```

To signal the get_nfn function that you want the first file name, you must set the most significant bit of the first byte, the disk number.

Here is an example showing how to use the get_nfn function:

```
struct _fcb fcb;          /* Declare a file control block */

setmem(fcb,FCB_SIZE,0);   /* Clear FCB to zeros */
fcb.fcb_disk = 0x80;      /* Mark FCB for "first time" */

while (get_nfn(fcb,"B:XYZ*.*"))
                          /* Until get_nfn returns a zero */
     {
                          /* Open the file using FCB */
     while               (/* Not at end of file */)
          {
                          /* Process next record or
                             Character in file*/
          }
                          /* Close the file */
     }
```

The quoted string "B:XYZ*.*" could also be just a pointer to a string, or a parameter on the command line, argv[n].

The last function for BDOS processing of the file directory, conv_fname, is used to convert a file name for output to a terminal. Again, the repetitive code at the end clears the file attribute bits to avoid any side effects from the terminal.

Utility Programs Enhancing Standard CP/M

This group of utilities is designed to enhance those supplied by Digital Research. They do not take advantage of any special features of the enhanced BIOS in Figure 8-10 and can be used on *any* CP/M Version 2.2 installation.

With the exception of the ERASE utility, all of the utilities scan down the file directory using BIOS calls, as described earlier in this chapter.

ERASE — A Safer Way to Erase Files

There are two disadvantages to the Console Command Processor's built-in ERA command. First, it will unquestioningly erase groups of files. Second, if you have a file name with nongraphic or lowercase characters, you cannot use the ERA command, as the CCP converts the command tail characters to uppercase and terminates a file name on encountering any strange character in the string.

The ERASE utility shown in Figure 11-3 erases groups of files, but it asks the user for confirmation before it erases each file.

Rather than use the BIOS to access each directory entry, it uses the get_nfn function, which then calls the BDOS. Thus ERASE functions equally well for files that have multiple entries in the directory. It can use the BDOS Delete File function to erase all extents of a given file.

Here is an example console dialog showing ERASE in operation:

```
P3A>erase<CR>
ERASE Version 1.0 02/23/83 (Library 1.0)
Usage :
        ERASE {d:}file_name.typ


P3A>erase *.com<CR>
ERASE Version 1.0 02/23/83 (Library 1.0)

Searching for file(s) matching A:????????.COM.
        Erase A:UNERASE .COM y/n? n
        Erase A:TEMP1   .COM y/n? y <== Will be Erased!
        Erase A:TEMP2   .COM y/n? n
        Erase A:TEMP3   .COM y/n? n
        Erase A:TEMP4   .COM y/n? y <== Will be Erased!
        Erase A:ERASE   .COM y/n? n

Erasing files now...
        File A:TEMP1   .COM erased.
        File A:TEMP4   .COM erased.
```

```
#define VN "1.0 02/24/83"

/* ERASE
   This utility erases the specified file(s) logically
   by using a BDOS delete function. */

#include <LIBRARY.H>

struct _fcb amb_fcb;            /* Ambiguous name file control block */
struct _fcb fcb;                /* Used for BDOS search functions */

char file_name[20];             /* Formatted for display: d:FILENAME.TYP */
short cur_disk;                 /* Current logical disk at start of program */
                                /* ERASE saves the FCB's of the all the
                                   files that need to be erased in the
                                   following array  */
#define MAXERA 1024
struct _fcb era_fcb[MAXERA];
int ecount;                     /* Count of number of files to be erased */
int count;                      /* Used to access era_fcb during erasing */


main(argc,argv)
short argc;             /* Argument count */
char *argv[];           /* Argument vector (pointer to an array of char .   */
{

printf("\nERASE Version %s (Library %s)",VN,LIBVN);
chk_use(argc);                  /* Check usage */
cur_disk = bdos(GETDISK);       /* Get current default disk */

ecount = 0;                     /* Initialize count of files to erase */

setfcb(amb_fcb,argv[1]);        /* Set ambiguous file name */
if (amb_fcb.fcb_disk)           /* Check if default disk to be used */
        {
        bdos(SETDISK,amb_fcb.fcb_disk + 1);     /* Set to specified disk */
        }
```

Figure 11-3. ERASE.C, a utility that requests confirmation before erasing

```
        /* Convert ambiguous file name for output */
conv_fname(amb_fcb,file_name);
printf("\n\nSearching for file(s) matching %s.",file_name);

        /* Set the file control block to indicate a "first" search */
fcb.fcb_disk |= 0x80;   /* OR in the ms bit */

        /* While not at the end of the directory, set the FCB
           to the next name that matches */
while(get_nfn(amb_fcb,fcb))
        {
        conv_fname(fcb,file_name);
                /* Ask whether to erase file or not */
        printf("\n\tErase %s y/n? ",file_name);
        if (toupper(getchar()) == 'Y')
                {
                printf(" <== Will be erased!");
                        /* add current fcb to array of FCB's */
                movmem(fcb,&era_fcb[ecount++],FCBSIZE);
                        /* Check that the table is not full */
                if (ecount == MAXERA)
                        {
                        printf("\nWarning : Internal table now full. No more files can be erased");
                        printf("\n   until those already specified have been erased.");
                        break;  /* Break out of while loop */
                        }
                }
        }       /* All directory entries processed */

if (ecount)
        printf("\n\nErasing files now...");

        /* now process each FCB in the array, erasing the files */
for (count = 0;         /* Starting with the first file in the array */
     count < ecount;    /* Until all active entries processed */
     count++)           /* Move to next FCB */
        {
        conv_fname(&era_fcb[count],file_name);
        if (bdos(DELETEF,&era_fcb[count]) == -1)        /* error? */
                printf("\n\007Error trying to erase %s",file_name);
        else            /* File erased */
                printf("\n\tFile %s erased.",file_name);
        }
bdos(SETDISK,cur_disk); /* reset to current disk */
}


chk_use(argc)           /* Check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions if not. */


/* Entry parameter */
int argc;       /* Count of the number of arguments on the command line */

{
        /* The minimum value of argc is 1 (for the program name itself),
           so argc is always one greater than the number of parameters
           on the command line */

if (argc != 2)
        {
        printf("\nUsage :");
        printf("\n\tERASE {d:}file_name.typ");
        exit();
        }

}
```

Figure 11-3. (Continued)

UNERASE — Restore Erased Files

UNERASE, as its name implies, can be used to "revive" an accidentally erased file. Only files whose allocation blocks have not been reallocated to other files can be revived. The UNERASE utility shown in Figure 11-4 builds a bit vector of all the allocation blocks used by active directory entries. Then it builds a bit vector for all the allocation blocks required by the file to be UNERASEd. If a Boolean AND between the two vectors yields a nonzero vector, then one or more blocks that originally belonged to the erased file are now allocated to other files on the disk.

```
#define VN "1.0 02/12/83"

/* UNERASE --
   This utility does the inverse of ERASE: it restores
   specified files to the directory by changing the first byte of
   their directory entries from 0xE5 back to the specified user
   number. */

#include <LIBRARY.H>

struct _dirpb dir_pb;          /* Directory management parameter block */
struct _dir *dir_entry;        /* Pointer to directory entry */
struct _scb scb;               /* Search control block */
struct _scb scba;              /* SCB set up to match all files */
struct _dpb dpb;               /* CP/M's disk parameter block */
struct _bv inuse_bv;           /* Bit vector for blocks in use */
struct _bv file_bv;            /* Bit vector for file to be unerased */
struct _bv extents;            /* Bit vector for those extents unerased */

char file_name[20];            /* Formatted for display : un/d:FILENAME.TYP */

short cur_disk;                /* Current logical disk at start of program
                                     NZ = show map of number of files */
int count;                     /* Used to access the allocation block numbers
                                     in each directory entry */
int user;                      /* User in which the file is to be revived */


main(argc,argv)
short argc;             /* Argument count */
char *argv[];           /* Argument vector (pointer to an array of chars.) */

{
printf("\nUNERASE Version %s (Library %s)",VN,LIBVN);
chk_use(argc);                  /* Check usage */
cur_disk = bdos(GETDISK);       /* Get current default disk */

        /* Using a special version of the set search-control-block utility,
           set the disk, name, type (no ambiguous names), the user number
           to match only erased entries, and the length to compare
           the user, name, and type.
           This special version also returns the disk_id taken from
           the file name on the command line.  */
if ((dir_pb.dp_disk = ssetscb(scb,argv[1],0xE5,12)) == 0)
        {       /* Use default disk */
        dir_pb.dp_disk = cur_disk;
        }
else
        {       /* make disk A = 0, B = 1 (for SELDSK) */
        dir_pb.dp_disk--;
        }
printf("\nSearching disk %d.",dir_pb.dp_disk);

if(strscn(scb,"?"))     /* Check if ambiguous name */
        {
        printf("\nError -- UNERASE can only revive a single file at a time.");
        exit();
```

Figure 11-4. UNERASE.C, a utility program that "revives" erased files

```
        }

        /* Set up a special search control block that will match with
           all existing files. */

ssetscb(scba,"*.*",'?',12);     /* Set file name and initialize SCB */

if (argc == 2)                  /* No user number specified */
        user = bdos(GETUSER,0xFF);      /* Get current user number */
else
{
        user = atoi(argv[2]);           /* Get specified number */
        if (user > 15)
                {
                printf("\nUser number can only be 0 - 15.");
                exit();
                }
}

/* Build a bit vector that shows the allocation blocks
   currently in use. SCBA has been set up to match all
   active directory entries on the disk. */
build_bv(inuse_bv,scba);

/* Build a bit vector for the file to be restored showing
   which allocation blocks will be needed for the file. */
if (!build_bv(file_bv,scb))
        {
        printf("\nNo directory entries found for file %s.",
                argv[1]);
        exit();
        }
/* Perform a boolean AND of the two bit vectors. */
bv_and(file_bv,inuse_bv,file_bv);

/* Check if the result is nonzero -- if so, then one or more
   of the allocation blocks required by the erased file is
   already in use for an existing file and the file cannot
   be restored. */
if (bv_nz(file_bv))
{
        printf("\n--- This file cannot be restored as some parts of it");
        printf("\n    have been re-used for other files! ---");
        exit();
}


/* Continue on to restore the file by changing all the entries
   in the directory to have the specified user number.
   Note: There may be several entries in the directory for
   the same file name and type, and even with the same extent
   number. For this reason, a bit map is kept of the extent
   numbers unerased -- duplicate extent numbers will not be
   unerased. */

/* Set up the bit vector for up to 127 unerased extents */
bv_make(extents,16);            /* 16 * 8 bits */

/* Set the directory to "closed", and force the get_nde
   function to open it. */
dir_pb.dp_open = 0;

/* While not at the end of the directory, return a pointer to
   the next entry in the directory. */
while(dir_entry = get_nde(dir_pb))
{

        /* Check if user = 0xE5 and name, type match */
if (comp_fname(scb,dir_entry) == NAME_EQ)
        {
                /* Test if this extent has already been
                   unerased */
        if (bv_test(extents,dir_entry -> de_extent))
                {               /* Yes it has */
                printf("\n\t\tExtent #%d of %s ignored.",
                        dir_entry -> de_extent,argv[1]);
                continue;       /* Do not unerase this one */
                }
```

Figure 11-4. (Continued)

```
        else                    /* Indicate this extent unerased */
                {
                bv_set(extents,dir_entry -> de_extent);
                dir_entry -> de_userno = user; /* Unerase entry */
                dir_pb.dp_write = 1;    /* Need to write sector back */
                printf("\n\tExtent #%d of %s unerased.",
                        dir_entry -> de_extent,argv[1]);
                }
        }
}

printf("\n\nFile %s unerased in User Number %d.",
argv[1],user);

bdos(SETDISK,cur_disk); /* Reset to current disk */
}


build_bv(bv,scb)        /* Build bit vector (from directory) */
/* This function scans the directory of the disk specified in
   the directory parameter block (declared as a global variable),
   and builds the specified bit vector, showing all the allocation
   blocks used by files matching the name in the search control
   block. */

/* Entry parameters */
struct _bv *bv;         /* Pointer to the bit vector */
struct _scb *scb;       /* Pointer to search control block */
/* Also uses : directory parameter block (dir_pb) */

/* Exit parameters
   The specified bit vector will be created, and will have 1-bits
   set wherever an allocation block is found in a directory
   entry that matches the search control block.
   It also returns the number of directory entries matched. */
{
unsigned abno;          /* Allocation block number */
struct _dpb *dpb;       /* Pointer to the disk parameter block in the BIOS */
int mcount;             /* Match count of dir. entries matched */

mcount = 0;             /* Initialize match count */
dpb = get_dpb(dir_pb.dp_disk);  /* Get disk parameter block address */

/* make the bit vector with one byte for each eight allocation
   blocks + 1 */
if (!(bv_make(bv,(dpb -> dpb_maxabn >>3)+1)))
        {
        printf("\nError -- Insufficient memory to make a bit vector.");
        exit();
        }


/* Set directory to "closed" to force the get_nde
   function to open it. */
dir_pb.dp_open = 0;

/* Now scan the directory building the bit vector */
while(dir_entry = get_nde(dir_pb))
        {
                /* Compare user number (which can legitimately be
                   0xE5), the file name and the type). */
        if (comp_fname(scb,dir_entry) == NAME_EQ)
                {
                ++mcount;               /* Update match count */
                for (count = 0;         /* Start with the first alloc. block */
                    count < dir_pb.dp_nabpde;   /* For number of alloc. blks. per dir. entry */
                    count++)
                        {
                                /* Set the appropriate bit number for
                                   each nonzero allocation block number */
                        if (dir_pb.dp_nabpde == 8)      /* assume 8 2-byte numbers */
                                {
                                abno = dir_entry -> _dirab.de_long[count];
                                }
                        else    /* Assume 16 1-byte numbers */
                                {
```

Figure 11-4. (Continued)

```
                                        abno = dir_entry -> _dirab.de_short[count];
                                        }
                                if (abno) bv_set(bv,abno); /* Set the bit */
                                }
                        }
                }
return mcount;                  /* Return number of dir. entries matched */
}


chk_use(argc)                   /* Check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions
   if not. */

/* Entry parameter */
int argc;               /* Count of the number of arguments on the command line */
{


/* The minimum value of argc is 1 (for the program name itself),
   so argc is always one greater than the number of parameters
   on the command line */

if (argc == 1 || argc > 3)
        {
        printf("\nUsage :");
        printf("\n\tUNERASE {d:}filename.typ {user}");
                printf("\n\tOnly a single unambiguous file name can be used.)");
        exit();
        }
} /* end chk_use */


ssetscb(scb,fname,user,length)  /* Special version of set search control block */
/* This function sets up a search control block according
   to the file name, type, user number, and number of bytes
   to compare.
   The file name can take the following forms :
        filename
        filename.typ
        d:filename.typ

   It sets the bit map according to which disks should be searched.
   For each selected disk, it checks to see if an error is generated
   when selecting the disk (i.e. if there are disk tables in the BIOS
   for the disk). */

/* Entry parameters */
struct _scb *scb;               /* Pointer to search control block */
char *fname;                    /* Pointer to the file name */
short user;                     /* User number to be matched */
int length;                     /* Number of bytes to compare */

/* Exit parameters
   Disk number to be searched. (A = 1, B = 2...)
*/
{
short disk_id;                  /* Disk number to search */

setfcb(scb,fname);              /* Set search control block as though it
                                   were a file control block. */
disk_id = scb -> scb_userno;            /* Set disk_id before it gets overwritten
                                           by the user number */
scb -> scb_userno = user;               /* Set user number */
scb -> scb_length = length;             /* Set number of bytes to compare */
return disk_id;
} /* end setscb */
```

Figure 11-4. (Continued)

A further complication occurs if two or more directory entries of the erased file have the same extent number. This can happen if the file has been created and erased several times. Under these circumstances, UNERASE revives the first entry with a given extent number that it encounters, and displays a message on the console both when an extent is revived and when one is ignored.

Because of the complicated nature of the UNERASE process, the utility can process only a single, unambiguous file name.

The following console dialog shows UNERASE in operation:

```
P3A>dir *.com<CR>
A: UNERASE  COM : TEMP2    COM : TEMP3    COM : ERASE    COM

P3A>unerase<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Usage :
        UNERASE {d:}filename.typ {user}
        Only a single unambiguous file name can be used.

P3A>unerase temp1.com<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Searching disk A.
        Extent #0 of TEMP1.COM unerased.
                Extent #0 of TEMP1.COM ignored.

File TEMP1.COM unerased in User Number 3.

P3A>dir *.com<CR>
A: UNERASE  COM : TEMP1    COM : TEMP2    COM : TEMP3    COM
A: ERASE    COM

P3A>unerase temp5.com<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Searching disk A.
No directory entries found for file TEMP5.COM.
```

FIND — Find "Lost" Files

The FIND utility shown in Figure 11-5 searches all user numbers on specified logical disks, matching each entry against an ambiguous file name. It can then display either a disk map showing how many matching files were found in each user number for each disk, or the user number, file name, and type for each matched directory entry.

You can use FIND to locate a specific file or group of files, as shown in the following console dialog:

```
P3B>find<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
Usage :
        FIND d:filename.typ {NAMES}
             *:filename.typ (All disks)
             ABCD..OP:filename.typ (Selected Disks)
        NAMES option shows actual names rather than map.

P3B>find ab:*.*<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
```

```
Searching disk : A
Searching disk : B
                    Numbers show files in each User Number.
                          --- User Numbers ---          Dir. Entries
       0    1    2    3    4    5   ...   11   12   13   14   15  Used Free
A:     1    1         8                                              23  233
B:    66   20   74   55    3                                        252  772

P3B>find *:*.com<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
Searching disk : A
Searching disk : B
Searching disk : C
                          --- User Numbers ---          Dir. Entries
       0    1    2    3    4    5   ...   11   12   13   14   15  Used Free
A:                    5                                              23  233
B:    61    5    4   13                                             252  772
C:   -- None --                                                      16  112

P3B>find *.com names<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
Searching disk : B
 0/B:CC      .COM 0/B:CC2     .COM 0/B:CLINK   .COM 2/B:CLIB    .COM
 1/B:CPM61   .COM 1/B:MOVCPM  .COM 1/B:PSWX    .COM 0/B:SUBMIT  .COM
 2/B:CDB     .COM 1/B:CPM60   .COM 0/B:DDT     .COM 0/B:EREMOTE .COM
 0/B:SPEEDSP .COM 0/B:PIP     .COM 0/B:PROTOSP .COM 0/B:RX      .COM
 0/B:TXA     .COM 0/B:EPUB    .COM 0/B:EPRIV   .COM 0/B:WSC     .COM
 0/B:X       .COM 0/B:CRCK    .COM 0/B:XSUB    .COM 0/B:DU      .COM
 0/B:QERA    .COM 0/B:FINDALL .COM 0/B:MOVEF   .COM 0/B:REMOTE  .COM
 0/B:LOCAL   .COM 0/B:DUMP    .COM 0/B:MRESET  .COM 0/B:ELOCAL  .COM
 0/B:PUTCPMF5.COM 0/B:TEST    .COM 0/B:FDUMP   .COM 0/B:INVIS   .COM
 0/B:L80     .COM 0/B:LIST    .COM 0/B:PUB     .COM 0/B:LOAD    .COM
 0/B:MAC     .COM 0/B:SCRUB   .COM 0/B:RXA     .COM 0/B:STAT    .COM
 0/B:TX      .COM 0/B:ERASEALL.COM 0/B:WM      .COM 0/B:MSFORMAT.COM
 0/B:STATUS  .COM 0/B:UNERA   .COM 0/B:MSINIT  .COM 0/B:VIS     .COM
 0/B:WSVTIP  .COM 0/B:XD      .COM 0/B:NEWVE   .COM 0/B:DDUMP   .COM
 0/B:FORMATMA.COM 0/B:PRIV    .COM 0/B:FCOMP   .COM 0/B:DDUMPA  .COM
 0/B:PUTSYS1C.COM 0/B:DDUMPNI .COM 0/B:DSTAT   .COM 0/B:ASM     .COM
 2/B:CDBTEST .COM 0/B:OLDSYS  .COM 0/B:E       .COM 2/B:F/C     .COM
 3/B:ERASE   .COM 3/B:FUNKEY  .COM 3/B:DATE    .COM 3/B:FIND    .COM

Press Space Bar to continue....
 3/B:SPACE   .COM 3/B:UNERASE .COM 3/B:MAKE    .COM 3/B:MOVE    .COM
 1/B:PUTSYSWX.COM 3/B:TIME    .COM 3/B:ASSIGN  .COM 3/B:SPEED   .COM
 3/B:PROTOCOL.COM 0/B:PRINTC  .COM 3/B:T       .COM
```

```
#define VN "1.0 02/11/83"

/* FIND - This utility can display either a map showing on which disks
   and in which user numbers files matching the specified ambiguous
   file name are found, or the actual names matched. */

#include <LIBRARY.H>

struct _dirpb dir_pb;           /* Directory management parameter block */
struct _dir *dir_entry;         /* Pointer to directory entry (somewhere in
                                      dir_pb) */
struct _scb scb;                /* Search control block */

char file_name[20];             /* Formatted for display : un/d:FILENAME.TYP */
```

Figure 11-5. FIND.C, a utility program that locates specific files or groups of files

```
short cur_disk;               /* Current logical disk at start of program */
int mcount;                   /* Match count (no. of file names matched) */
int dmcount;                  /* Per disk match count */
int lcount;                   /* Line count (for lines displayed) */

int map_flag;                 /* 0 = show file names of matched files,
                                 NZ = show map of number of files */

        /* The array below is used to tabulate the results for each
           disk drive, and for each user number on the drive.
           In addition, two extra "users" have been added for "free"
           and "used" values. */

unsigned disk_map[16][18];    /* Disk A -> P, users 0 -> 15, free, used */
#define USED_COUNT 16         /* "User" number for used entities */
#define FREE_COUNT 17         /* "User" number for free entities */


main(argc,argv)
short argc;             /* Argument count */
char *argv[];           /* Argument vector (pointer to an array of chars.) */
{

printf("\nFIND Version %s (Library %s)",VN,LIBVN);
chk_use(argc);                /* Check usage */
cur_disk = bdos(GETDISK);     /* Get current default disk */

dm_clr(disk_map);             /* Reset disk map */

        /* Set search control block
           disks, name, type, user number, extent number,
           and number of bytes to compare -- in this case, match all users,
           but only extent 0 */
setscb(scb,argv[1],'?',0,13);  /* Set disks, name, type */

map_flag = usstrcmp("NAMES",argv[2]);  /* Set flag for map option */

lcount = dmcount = mcount = 0;         /* Initialize counts */

for (scb.scb_disk = 0;        /* Starting with logical disk A: */
     scb.scb_disk < 16;       /* Until logical disk P: */
     scb.scb_disk++)          /* Move to next logical disk */
{

        /* Check if current disk has been selected for search */
if (!(scb.scb_adisks & (1 << scb.scb_disk)))
        continue;       /* No,so bypass this disk */

printf("\nSearching disk : %c",(scb.scb_disk + 'A'));
lcount++;               /* Update line count */

dir_pb.dp_disk = scb.scb_disk;  /* Set to disk to be searched*/
dmcount = 0;                    /* Reset disk matched count */

if (!map_flag)          /* If file names are to be displayed */
        putchar('\n');  /* Move to column 1 */

/* Set the directory to "closed", and force the get_nde
   function to open it  */
dir_pb.dp_open = 0;

        /* While not at the end of the directory, set a pointer to the
           next directory entry  */
while(dir_entry = get_nde(dir_pb))
        {
        /* Check if entry in use, to update
           the free/used counts */

        if (dir_entry -> de_userno == 0xE5)     /* Unused */
                disk_map[scb.scb_disk][FREE_COUNT]++;
        else    /* In use */
                disk_map[scb.scb_disk][USED_COUNT]++;

        /* Select only those active entries that are the
           first extent (numbered 0) of a file that matches
           the name supplied by the user  */
```

Figure 11-5. (Continued)

```
        if (
            (dir_entry -> de_userno != 0xE5) &&
            (dir_entry -> de_extent == 0) &&
            (comp_fname(scb,dir_entry) == NAME_EQ)
           )
                {

                mcount++;       /* Update matched counts */
                dmcount++;      /* Per disk count */

                if (map_flag)   /* Check map option */
                        {
                                /* Update disk map */
                        disk_map[scb.scb_disk][dir_entry -> de_userno]++;
                        }
                else            /* Display names */
                        {
                        conv_dfname(scb.scb_disk,dir_entry,file_name);
                        printf("%s  ",file_name);

                                /* Check if need to start new line */
                        if (!(dmcount % 4))
                                {
                                putchar('\n');
                                        if (++lcount > 18)
                                                {
                                                lcount = 0;
                                                printf("\nPress Space Bar to continue....");
                                                getchar();
                                                putchar('\n');
                                                }
                                        }
                                }
                        }
                } /* End of directory */
        } /* All disks searched */

if (map_flag)
{
printf("\n                Numbers show files in each user number.");
printf("\n                          --- User Numbers ---                     Dir. Entries");

dm_disp(disk_map,scb.scb_adisks);       /* Display disk map */
}

if (mcount == 0)
printf("\n --- File Not Found --- ");

bdos(SETDISK,cur_disk); /* Reset to current disk */
}


chk_use(argc)           /* check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions
   if not.
*/

/* Entry parameter */
int argc;       /* Count of the number of arguments on the command line */
{

/* The minimum value of argc is 1 (for the program name itself),
   so argc is always one greater than the number of parameters
   on the command line */

if (argc == 1 || argc > 3)
{
printf("\nUsage :");
printf("\n\tFIND d:filename.typ {NAMES}");
printf("\n\t     *:filename.typ (All disks)");
printf("\n\t     ABCD..OP:filename.typ (Selected Disks)");
printf("\n\tNAMES option shows actual names rather than map.");
exit();
}

}
```

Figure 11-5. (Continued)

SPACE — Show Used Disk Space

The SPACE utility shown in Figure 11-6 scans the specified logical disks and displays a disk map that shows, for each user number on each logical disk, how many Kbytes of storage have been used. It also displays the total number of Kbytes used and free on each logical disk.

Here is an example console dialog showing SPACE in operation:

```
P3B>space<CR>
SPACE Version 1.0 02/11/83 (Library 1.0)
Usage :
        SPACE *        (All disks)
        SPACE ABCD..OP (Selected Disks)

P3B>space *<CR>
SPACE Version 1.0 02/11/83 (Library 1.0)
Searching disk : A
Searching disk : B
Searching disk : C
                  Numbers show space used in kilobytes.
                          --- User Numbers ---              Space (Kb)
    0   1   2   3   4   5  ...   10  11  12  13  14  15  Used Free
A:  18 202      38                                         258 1196
B: 692 432 656 548  36                                    2364  996
C: 140                                                     140  204
```

```
#define VN "1.0 02/11/83"

/* SPACE -- This utility displays a map showing on the amount of space
   (expressed as relative percentages) occupied in each user number
   for each logical disk.It also shows the relative amount of space
   free. */

#include <LIBRARY.H>

struct _dirpb dir_pb;           /* Directory management parameter block */
struct _dir *dir_entry;         /* Pointer to directory entry */
struct _scb scb;                /* Search control block */
struct _dpb dpb;                /* CP/M's disk parameter block */

char file_name[20];             /* Formatted for display : un/d:FILENAME.TYP */

short cur_disk;         /* Current logical disk at start of program
                           NZ = show map of number of files */
int count;              /* Used to access the allocation block numbers
                           in each directory entry */
int user;               /* Used to access the disk map when calculating */

/* The array below is used to tabulate the results for each
   disk drive, and for each user number on the drive.
   In addition, two extra "users" have been added for "free"
   and "used" values.
*/
unsigned disk_map[16][18];      /* Disk A -> P, users 0 -> 15, free, used */
#define USED_COUNT 16           /* "User" number for used entities */
#define FREE_COUNT 17           /* "User" number for free entities */


main(argc,argv)
short argc;             /* Argument count */
char *argv[];           /* Argument vector (pointer to an array of chars.) */
{
```

Figure 11-6. SPACE.C, a utility that displays how much disk storage is used or available

```
  printf("\nSPACE Version %s (Library %s)",VN,LIBVN);
  chk_use(argc);                    /* Check usage */
  cur_disk = bdos(GETDISK);         /* Get current default disk */

  dm_clr(disk_map);                 /* Reset disk map */

  ssetscb(scb,argv[1]);             /* Special version : set disks,
                                       name, type */

  for (scb.scb_disk = 0;            /* Starting with logical disk A: */
       scb.scb_disk < 16;           /* Until logical disk P: */
       scb.scb_disk++)              /* Move to next logical disk */
          {

          /* Check if current disk has been selected for search */
          if (!(scb.scb_adisks & (1 << scb.scb_disk)))
                  continue;         /* No, so bypass this disk */

          printf("\nSearching disk : %c",(scb.scb_disk + 'A'));
          dir_pb.dp_disk = scb.scb_disk;  /* Set to disk to be searched */

          /* Set the directory to "closed", and force the get_nde
             function to open it  */
          dir_pb.dp_open = 0;

          /* While not at the end of the directory, set a pointer
             to the next entry in the directory */
          while (dir_entry = get_nde(dir_pb))
                  {
                  if (dir_entry -> de_userno == 0xE5)
                          continue;         /* Bypass inactive entries */

                  for (count = 0;         /* Start with the first alloc. block */
                       count < dir_pb.dp_nabpde;  /* For number of alloc. blks. per dir. entry */
                       count++)
                          {
                          if (dir_pb.dp_nabpde == 8)      /* Assume 8 2-byte numbers */
                                  {
                                  disk_map[scb.scb_disk][dir_entry -> de_userno]
                                          += (dir_entry -> _dirab.de_long[count] > 0 ? 1 : 0);
                                  }
                          else    /* Assume 16 1-byte numbers */
                                  {
                                  disk_map[scb.scb_disk][dir_entry -> de_userno]
                                          += (dir_entry -> _dirab.de_short[count] > 0 ? 1 : 0);
                                  }
                          }       /* All allocation blocks processed */
                  }       /* End of directory for this disk */


          /* Compute the storage used by multiplying the number of
             allocation blocks counted by the number of Kbytes in
             each allocation block. */

          for (user = 0;  /* Start with user 0 */
               user < 16; /* End with user 15 */
               user ++)   /* Move to next user number */
                  {
                          /* Compute size occupied in Kbytes */
                  disk_map[scb.scb_disk][user] *= dir_pb.dp_absize;
                          /* Build up sum for this disk */
                  disk_map[scb.scb_disk][USED_COUNT] += disk_map[scb.scb_disk][user];
                  }

          /* Free space = (# of alloc. blks * # of kbyte per blk)
                  - used Kbytes
                  - (directory entries * 32) / 1024 ... or divide by 32 */
          disk_map[scb.scb_disk][FREE_COUNT] = (dir_pb.dp_nab * dir_pb.dp_absize)
                  - disk_map[scb.scb_disk][USED_COUNT]
                  - (dir_pb.dp_nument >> 5);      /* Same as / 32 */
          }       /* All disks processed */


 printf("\n                    Numbers show space used in kilobytes.");
 printf("\n                             --- User Numbers ---                  Space (Kb)");

 dm_disp(disk_map,scb.scb_adisks);        /* Display disk map */
```

Figure 11-6. (Continued)

```
bdos(SETDISK,cur_disk); /* Reset to current disk */
}

ssetscb(scb,ldisks)     /* Special version of set search control block */

/* This function sets up a search control block according
   to just the logical disks specified. The disk are specified as
   a single string of characters without any separators. An
   asterisk means "all disks." For example --

        ABGH    (disks A:, B:, G: and H: )
        *       (all disks for which SELDSK has tables)

   It sets the bit map according to which disks should be searched.
   For each selected disk, it checks to see if an error is generated
   when selecting the disk (i.e. if there are disk tables in the BIOS
   for the disk).
   The file name, type, and extent number are all set to "?" to match
   all possible entries in the directory. */

/* Entry parameters */
struct _scb *scb;       /* Pointer to search control block */
char *ldisks;           /* Pointer to the logical disks */

/* Exit parameters
   None.
*/
{
int disk;               /* Disk number currently being checked */
unsigned adisks;        /* Bit map for active disks */

adisks = 0;             /* Assume no disks to search */

if (*ldisks)            /* Some values specified */
        {
        if (*ldisks == '*')     /* Check if "all disks" */
                {
                adisks = 0xFFFF;        /* Set all bits */
                }
        else                    /* Set specific disks */
                {
                while(*ldisks)  /* Until end of disks reached */
                        {
                        /* Build the bit map by getting the next disk
                           id. (A - P), converting it to a number
                           in the range 0 - 15, and shifting a 1-bit
                           left that many places and OR ing it into
                           the current active disks.
                        */
                        adisks |= 1 << (toupper(*ldisks) - 'A');
                        ++ldisks;       /* Move to next character */
                        }
                }
        }
else    /* Use only current default disk */
        {
        /* Set just the bit corresponding to the current disk */
        adisks = 1 << bdos(GETDISK);
        }

        /* Set the user number, file name, type, and extent to "?"
           so that all active directory entries will match */
                /*      0123456789012          */
strcpy(&scb -> scb_userno,"?????????????");

        /* Make calls to the BIOS SELDSK routine to make sure that
           all of the active disk drives have disk tables for them
           in the BIOS. If they don't, turn off the corresponding
           bits in the bit map. */


for (disk = 0;          /* Start with disk A: */
     disk < 16;         /* Until disk P: */
     disk++)            /* Use next disk */
        {
        if ( !((1 << disk) & adisks))
                continue;               /* Avoid selecting unspecified disks */
```

Figure 11-6. (Continued)

```
        if (biosh(SELDSK,disk) == 0)    /* Make BIOS SELDSK call */
                {                       /* Returns 0 if invalid disk */
                /* Turn OFF corresponding bit in mask
                   by AND-ing it with bit mask having
                   all the other bits set = 1. */
                adisks &= ((1 << disk) ^ 0xFFFF);
                }
        }

scb -> scb_adisks = adisks;     /* Set bit map in scb */

} /* End ssetscb */



chk_use(argc)           /* Check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions
   if not. */

/* Entry parameter */
int argc;       /* Count of the number of arguments on the command line */
{

        /* The minimum value of argc is 1 (for the program name itself),
           so argc is always one greater than the number of parameters
           on the command line */

if (argc != 2)
        {
        printf("\nUsage :");
        printf("\n\tSPACE *        (All disks)");
        printf("\n\tSPACE ABCD..OP (Selected Disks)");
        exit();
        }
} /* End chk_use */
```

Figure 11-6. (Continued)

MOVE — Move Files Between User Numbers

The MOVE utility shown in Figure 11-7 moves files from one user number to another on the same logical disk. The movement is achieved by changing the user number in all the relevant directory entries. This is much faster than copying the files. It also avoids having multiple copies of the same file on the disk.

Here is a console dialog showing MOVE in operation:

```
P3B>move<CR>
MOVE Version 1.0 02/10/83 (Library 1.0)
Usage :
        MOVE d:filename.typ to_user {from_user} {NAMES}
             *:filename.typ (All disks)
             ABCD..OP:filename.typ (Selected Disks)
        NAMES option shows names of files moved.

P3B>dir *.com<CR>
B: ERASE    COM : FUNKEY  COM : DATE     COM : FIND     COM
B: SPACE    COM : UNERASE COM : MAKE     COM : MOVE     COM
B: TIME     COM : ASSIGN  COM : SPEED    COM : PROTOCOL COM

P3B>move *.com 0 names<CR>
MOVE Version 1.0 02/10/83 (Library 1.0)

Moving file(s)  3/B:????????.COM -> User 0.
```

```
0/B:ERASE   .COM  0/B:FUNKEY  .COM  0/B:DATE    .COM  0/B:FIND    .COM
0/B:SPACE   .COM  0/B:UNERASE .COM  0/B:MAKE    .COM  0/B:MOVE    .COM
0/B:TIME    .COM  0/B:ASSIGN  .COM  0/B:SPEED   .COM  0/B:PROTOCOL.COM

P3B>user 0<CR>
P0B>dir
B: ERASE    COM : FUNKEY  COM : DATE     COM : FIND     COM
B: SPACE    COM : UNERASE COM : MAKE     COM : MOVE     COM
B: TIME     COM : ASSIGN  COM : SPEED    COM : PROTOCOL COM
```

```
#define VN "1.0 02/10/83"

/* MOVE -- This utility transfers file(s) from one user number to
   another, but on the SAME logical disk. Files are not actually
   copied -- rather, their directory entries are changed. */

#include <LIBRARY.H>                    /

struct _dirpb dir_pb;           /* Directory management parameter block */
struct _dir *dir_entry;         /* Pointer to directory entry */
struct _scb scb;                /* Search control block */


#define DIR_BSZ 128             /* Directory buffer size */
char dir_buffer[DIR_BSZ];       /* Directory buffer */

char file_name[20];             /* Formatted for display : un/d:FILENAME.TYP */
short name_flag;                /* NZ to display names of files moved */

short cur_disk;                 /* Current logical disk at start of program */
int from_user;                  /* User number from which to move files */
int to_user;                    /* User number to which files will be moved */

int mcount;                     /* Match count (no. of file names matched) */
int dmcount;                    /* Per-disk match count */
int lcount;                     /* Line count (for lines displayed) */


main(argc,argv)
short argc;             /* Argument count */
char *argv[];           /* Argument vector (pointer to an array of chars.) */
{

printf("\nMOVE Version %s (Library %s)",VN,LIBVN);

chk_use(argc);                  /* Check usage */

to_user = atoi(argv[2]);        /* Convert user no. to integer */
        /* Set and check destination user number */
if(to_user > 15)
        {
        printf("\nError -- the destination user number cannot be greater than 15.");
        }

        /* Set the current user number */
from_user = bdos(GETUSER,0xFF);

        /* Check if source user number specified */
if (isdigit(argv[3][0]))
        {
                /* Set and check source user number */
        if((from_user = atoi(argv[3])) > 15)
                {
                printf("\nError -- the source user number cannot be greater than 15.");
                exit();
                }
                /* Set name suppress flag from parameter #4 */
        name_flag = usstrcmp("NAMES",argv[4]);
        }
else            /* No source user specified */
        {
```

Figure 11-7. MOVE.C, a utility program that changes files' user numbers

```
                /* Set name suppress flag from parameter #3 */
        name_flag = usstrcmp("NAMES",argv[3]);
        }

        /* To simplify the logic below, name_flag must be made
           NZ if it is equal to NAME_EQ, 0 if it is any other value */
name_flag = (name_flag == NAME_EQ ? 1 : 0);

if (to_user == from_user)       /* To = from */
        {
        printf("\nError - 'to' user number is the same as 'from' user number.");
        exit();
        }

        /* Set the search control block file name, type, user number,
           extent number, and length -- length matches user number, file
           name, and type. As the extent number does not enter into the
           comparison, all extents of a given file will be found. */
setscb(scb,argv[1],from_user,'?',13);

cur_disk = bdos(GETDISK);       /* Get current default disk */
lcount = dmcount = mcount = 0;  /* Initialize counts */

for (scb.scb_disk = 0;          /* Starting with logical disk A: */
     scb.scb_disk < 16;         /* Until logical disk P: */
     scb.scb_disk++)            /* Move to next logical disk */
        {
                /* Check if current disk has been selected for search */
        if (!(scb.scb_adisks & (1 << scb.scb_disk)))
                continue;       /* No, so bypass this disk */
                /* convert search user number and name for output */
        conv_dfname(scb.scb_disk,scb,file_name);
        printf("\n\nMoving file(s) %s -> User %d.",file_name,to_user);

        lcount++;               /* Update line count */

        dir_pb.dp_disk = scb.scb_disk;  /* Set to disk to be searched*/
        dmcount = 0;                    /* Reset disk matched count */

        if (name_flag)          /* If file names are to be displayed */
                putchar('\n');  /* Move to column 1 */

                /* Set the directory to "closed" to force the get_nde
                   function to open it. */
        dir_pb.dp_open = 0;

                /* While not at the end of the directory, set a pointer
                   to the next directory entry */
        while(dir_entry = get_nde(dir_pb))
                {
                        /* Match those entries that have the correct
                           user number, file name, type, and any
                           extent number. */
                if (
                    (dir_entry -> de_userno != 0xE5) &&
                    (comp_fname(scb,dir_entry) == NAME_EQ)
                   )
                        {

                        dir_entry -> de_userno = to_user;       /* Move to new user */
                                /* Request sector to be written back */
                        dir_pb.dp_write = 1;

                        mcount++;       /* Update matched counts */
                        dmcount++;      /* Per-disk count */

                        if (name_flag)  /* Check map option */
                                {
                                conv_dfname(scb.scb_disk,dir_entry,file_name);
                                        printf("%s  ",file_name);

                                /* Check if need to start new line */
                                if (!(dmcount % 4))
                                        {
                                        putchar('\n');
                                        if (++lcount > 18)
```

Figure 11-7. (Continued)

```
                                                            {
                                                            lcount = 0;
                                                            printf("\nPress Space Bar to continue....");
                                                            getchar();
                                                            putchar('\n');
                                                            }
                                                     }
                                              }
                                       }
                                }
                         }

        if (mcount == 0)
                printf("\n --- No Files Moved --- ");

        bdos(SETDISK,cur_disk); /* Reset to current disk */
        }


        chk_use(argc)           /* Check usage */
        /* This function checks that the correct number of
           parameters has been specified, outputting instructions
           if not  */
        /* Entry parameter */
        int argc;       /* Count of the number of arguments on the command line */
        {

        /* The minimum value of argc is 1 (for the program name itself),
           so argc is always one greater than the number of parameters
           on the command line */

        if (argc == 1 || argc > 5)
                {
                printf("\nUsage :");
                printf("\n\tMOVE d:filename.typ to_user {from_user} {NAMES}");
                printf("\n\t     *:filename.typ (All disks)");
                printf("\n\t     ABCD..OP:filename.typ (Selected Disks)");
                printf("\n\tNAMES option shows names of files moved.");
                exit();
                }

        }
```

Figure 11-7. (Continued)

Other Utilities

The utility programs described in this section are by no means a complete set. You may want to develop many other specialized utility programs. Some possibilities are:

FILECOPY
: A more specialized version of PIP could copy ambiguously specified groups of files. Of special importance would be the ability to read a file containing the names of the files to be copied. A useful option would be the ability to detect the setting of the unused file attribute bit and copy only files that have been changed.

PROTECT/UNPROTECT
: This pair of utilities would allow you to "hide" files in user numbers greater than 15. Files so hidden could not be accessed other than by UNPROTECTing them, thereby moving them back into the normal user number range.

RECLAIM
: This utility would read all sectors on a disk (using the BIOS). Any bad sectors encountered could then be logically removed by creating an entry in the file directory, with allocation block numbers that would effectively "reserve" the blocks containing the bad sectors.

OWNER
: This utility, given a track or sector number, would access the directory and determine which file or files were using that part of the disk. This is useful if you have a bad sector or track on a disk. You then can determine which files have been damaged.

Utility Programs for the Enhanced BIOS

This section describes several utility programs that work with the enhanced BIOS shown in Figure 8-10. Several of these utilities work directly with the physical devices on the computer system, which can vary from computer to computer. The library header contains #define declarations for device numbers and names for physical devices (Figure 11-2, f and Figure 11-2, g).

These #define statements are used to build a physical-device code table. If you have more physical devices or want to change the names by which you refer to the devices, you will need to change these definitions.

All of these utilities share some common features in the way that they are invoked. If they are called without any parameters, they display instructions on the console regarding what parameters are available. If they are called with the word "SHOW" (or "S", "SH", and so forth) as a parameter, they display the current settings of whatever attribute the utility controls.

MAKE — Make Files "Invisible" or "Visible"

The MAKE utility shown in Figure 11-8 is designed to operate in conjunction with the public files option implemented in the enhanced BIOS of Figure 8-10. It has two modes of operation — making files "invisible" or "visible."

An invisible file is one in user 0 which has been set to Read-Only and System status. When the public files option is enabled, these files cannot be seen when you use the DIR command, nor can they be erased accidentally.

A visible file is one that has been set to Read/Write and Directory status.

When files are made invisible, they are transferred from the current user number to user 0. When files are made visible, they are transferred from user 0 to the current user number.

Here is an example console dialog showing MAKE in operation:

```
P3B>make<CR>
MAKE Version 1.0 02/12/83 (Library 1.0)
```

```
Usage :
        MAKE d:filename.typ INVISIBLE {NAMES}
                            VISIBLE
             *:filename.typ (All disks)
             ABCD..OP:filename.typ (Selected Disks)
        NAMES option shows names of files processed.

P3B>dir *.com<CR>
B: ERASE    COM : UNERASE  COM : ASSIGN   COM : PROTOCOL COM

P3B>make *.com invisible names<CR>
MAKE Version 1.0 02/12/83 (Library 1.0)

Moving files from User 3 to 0 and making them Invisible.
Searching disk : B

         0/B:ERASE   .COM made Invisible in User 0.
         0/B:UNERASE .COM made Invisible in User 0.
         0/B:ASSIGN  .COM made Invisible in User 0.
         0/B:PROTOCOL.COM made Invisible in User 0.

P3B>make erase.com visible names<CR>
MAKE Version 1.0 02/12/83 (Library 1.0)

Moving files from User 0 to 3 and making them Visible.
Searching disk : B

         3/B:ERASE   .COM made Visible in User 3.
```

```
#define VN "1.0 02/12/83"

/* MAKE - This utility is really two very similar programs;
   which one depends on the parameter specified on the command
   line.

   INVISIBLE finds all of the specified files, moves them
   to user number 0, and sets them to be System and Read Only
   status. These files can then be accessed from user numbers
   other than 0 when the public files feature is enabled in the
   BIOS.

   VISIBLE is the opposite in that the specified files are
   moved to the current user number and changed to Directory
   and Read/Write status. */

#include <LIBRARY.H>

struct _dirpb dir_pb;          /* Directory management parameter block */
struct _dir *dir_entry;        /* Pointer to directory entry */
struct _scb scb;               /* Search control block */
short to_user;                 /* User number to which files will be set */
short from_user;               /* User number from which files will be moved */

char file_name[20];            /* Formatted for display : un/d:FILENAME.TYP */
short name_flag;               /* NZ to display names of files moved */

short cur_disk;                /* Current logical disk at start of program */

int mcount;                    /* Match count (no. of file names matched) */

short invisible;               /* NZ when parameter specifies invisible */
char *operation;               /* Pointer to either "invisible" or "visible" */

main(argc,argv)
short argc;               /* Argument count */
char *argv[];             /* Argument vector (pointer to an array of chars.) */
```

Figure 11-8. MAKE.C, a utility that makes files "invisible" and protected or makes them "visible," accessible, and unprotected

```
{

printf("\nMAKE Version %s (Library %s)",VN,LIBVN);
chk_use(argc);                  /* Check usage */
cur_disk = bdos(GETDISK);       /* Get current default disk */
mcount = 0;                     /* Initialize count */


        /* Set the invisible flag according to the parameter */
invisible = usstrcmp("VISIBLE",argv[2]);

        /* Set the from_user and to_user numbers depending on which
           program is to be built, and the parameters specified. */

if (invisible)
        {
        from_user = bdos(GETUSER,0xFF); /* Get current user number */
        to_user = 0;    /* Always move files to user 0 */
        operation = "Invisible";        /* Set pointer to string */
        }
else    /* visible */
        {
        from_user = 0;                  /* Always move from user 0 */
        to_user = bdos(GETUSER,0xFF);   /* Get current user */
        operation = "Visible";          /* Set pointer to string */
        }

        /* Set search control block disks, name, type, user number,
           extent number, and number of bytes to compare -- in this
           case, match the "from" user, all extents. */
setscb(scb,argv[1],from_user,'?',13);   /* Set disks, name, type */

name_flag = usstrcmp("NAMES",argv[3]);  /* Set name-suppress flag from param. 3 */

        /* To simplify the logic below, name_flag must be made
           NZ if it is equal to NAME_EQ, 0 if it is any other value */
name_flag = (name_flag == NAME_EQ ? 1 : 0);


        /* Convert search user number and name for output */
conv_dfname(scb.scb_disk,scb,file_name);
printf("\n\nMoving files from User %d to %d and making them %s.",
        from_user,to_user,operation);

for (scb.scb_disk = 0;          /* Starting with logical disk A: */
     scb.scb_disk < 16;         /* Until logical disk P: */
     scb.scb_disk++)            /* Move to next logical disk */
        {

                /* Check if current disk has been selected for search */
        if (!(scb.scb_adisks & (1 << scb.scb_disk)))
                continue;       /* No -- so bypass this disk */

        printf("\nSearching disk : %c",(scb.scb_disk + 'A'));

        dir_pb.dp_disk = scb.scb_disk;  /* Set to disk to be searched*/

        if (name_flag)          /* If file names are to be displayed */
                putchar('\n');  /* Move to column 1 */


                /* Set the directory to "closed", and force the get_nde
                   function to open it. */
        dir_pb.dp_open = 0;

                /* While not at the end of the directory,
                   set a pointer to the next directory entry. */
        while(dir_entry = get_nde(dir_pb))
                {

                        /* Match those entries that have the correct
                           user number, file name, type, and any
                           extent number. */
                if (
                    (dir_entry -> de_userno != 0xE5) &&
                    (comp_fname(scb,dir_entry) == NAME_EQ)
                   )
                        {
```

Figure 11-8. (Continued)

```
                                mcount++;       /* Update matched counts */

                                if (invisible)
                                        {       /* Set ms bits */
                                        dir_entry -> de_fname[8] |= 0x80;
                                        dir_entry -> de_fname[9] |= 0x80;
                                        }
                                else    /* Visible */
                                        {       /* Clear ms bits */
                                        dir_entry -> de_fname[8] &= 0x7F;
                                        dir_entry -> de_fname[9] &= 0x7F;
                                        }

                                        /* Move to correct user number */
                                dir_entry -> de_userno = to_user;

                                        /* Indicate sector to be written back */
                                dir_pb.dp_write = 1;

                                        /* Check if name to be displayed */
                                if (name_flag)
                                        {
                                        conv_dfname(scb.scb_disk,dir_entry,file_name);
                                        printf("\n\t%s made %s in User %d.",
                                                file_name,operation,to_user);
                                        }
                                }
                        }       /* All directory entries processed */
                }       /* All disks processed */

        if (mcount == 0)
                printf("\n --- No Files Processed --- ");

        bdos(SETDISK,cur_disk); /* Reset to current disk */
        }


chk_use(argc)           /* Check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions
   if not.
*/
/* Entry parameter */
int argc;       /* Count of the number of arguments on the command line */
{

        /* The minimum value of argc is 1 (for the program name itself),
           so argc is always one greater than the number of parameters
           on the command line */

if (argc == 3 || argc == 4)
        return;
else
        {
        printf("\nUsage :");
        printf("\n\tMAKE d:filename.typ INVISIBLE {NAMES}");
        printf("\n\t                    VISIBLE");
        printf("\n\t     *:filename.typ (All disks)");
        printf("\n\t     ABCD..OP:filename.typ (Selected Disks)");
        printf("\n\tNAMES option shows names of files processed.");
        exit();
        }

}
```

Figure 11-8. (Continued)

SPEED — Set Baud Rates

The SPEED utility shown in Figure 11-9 sets the baud rate for a specific serial device. Here is an example console dialog that shows several of the options:

```
P3B>speed<CR>
SPEED 1.0 02/17/83
The SPEED utility sets the baud rate speed for each physical device.
Usage is :  SPEED physical-device baud-rate, or
            SPEED SHOW     (to show current settings)

Valid physical devices are:
                TERMINAL
                PRINTER
                MODEM

Valid baud rates are:
                300
                600
                1200
                2400
                4800
                9600
                19200


P3B>speed show<CR>
SPEED 1.0 02/17/83
Current Baud Rate settings are :
        TERMINAL set to 9600 baud.
        PRINTER set to 9600 baud.
        MODEM set to 9600 baud.


P3B>speed m 19<CR>
SPEED 1.0 02/17/83
Current Baud Rate settings are :
        TERMINAL set to 9600 baud.
        PRINTER set to 9600 baud.
        MODEM set to 19200 baud.


P3B>speed xyz 12<CR>
SPEED 1.0 02/17/83
Physical Device 'XYZ' is invalid or ambiguous.
Legal Physical Devices are :
                TERMINAL
                PRINTER
                MODEM
```

```
#define VN "\nSPEED 1.0 02/17/83"

/* This utility sets the baud rate speed for each of the physical
   devices. */

#include <LIBRARY.H>

struct _ct ct_pdev[MAXPDEV + 2];        /* Physical device table */

        /* Hardware specific items */
```

Figure 11-9. SPEED.C, a utility that sets the baud rate for a specific device

```
                                   /* Baud rates for serial ports */
#define B300    0x35               /* 300 baud */
#define B600    0x36               /* 600 baud */
#define B1200   0x37               /* 1200 baud */
#define B2400   0x3A               /* 2400 baud */
#define B4800   0x3C               /* 4800 baud */
#define B9600   0x3E               /* 9600 baud */
#define B19200  0x3F               /* 19200 baud */
struct _ct ct_br[10];    /* Code table for baud rates (+ spare entries) */


        /* Parameters on the command line */
#define PDEV argv[1]     /* Physical device */
#define BAUD argv[2]     /* Baud rate */


main(argc,argv)
int argc;
char *argv[];
{
printf(VN);        /* Display sign-on message */
setup();           /* Set up code tables */
chk_use(argc);     /* Check correct usage */

        /* Check if request to show current settings */
if (usstrcmp("SHOW",argv[1]))
        {               /* No -- assume setting is required */
        set_baud(get_pdev(PDEV),get_baud(BAUD)); /* Set baud rate */
        }

show_baud();                /* Display current settings */

} /* end of program */

setup()                     /* set up the code tables for this program */
{
        /* Initialize the physical device table */
ct_init(ct_pdev[0],T_DEVN,PN_T);         /* Terminal */
ct_init(ct_pdev[1],P_DEVN,PN_P);         /* Printer */
ct_init(ct_pdev[2],M_DEVN,PN_M);         /* Modem */
ct_init(ct_pdev[3],CT_SNF,"*"); /* Terminator */

        /* Initialize the baud rate table */
ct_init(ct_br[0],B300,"300");
ct_init(ct_br[1],B600,"600");
ct_init(ct_br[2],B1200,"1200");
ct_init(ct_br[3],B2400,"2400");
ct_init(ct_br[4],B4800,"4800");
ct_init(ct_br[5],B9600,"9600");
ct_init(ct_br[6],B19200,"19200");
ct_init(ct_br[7],CT_SNF,"*");    /* Terminator */
}

unsigned
get_pdev(ppdev)             /* Get physical device */
/* This function returns the physical device code
   specified by the user in the command line. */
char *ppdev;                /* Pointer to character string */
{
unsigned retval;                          /* Return value */

retval = ct_parc(ct_pdev,ppdev);          /* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Physical Device '%s' is invalid or ambiguous.",
                ppdev);
        printf("\nLegal Physical Devices are : ");
        ct_disps(ct_pdev);      /* Display all values */
        exit();
        }
return retval;                  /* Return code */
}

unsigned
get_baud(pbaud)
/* This function returns the baud rate time constant for
   the baud rate specified by the user in the command line   */
```

Figure 11-9. (Continued)

```
char *pbaud;              /* Pointer to character string */
{
unsigned retval;                     /* Return value */
retval = ct_parc(ct_br,pbaud);  /* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Baud Rate '%s' is invalid or ambiguous.",
                pbaud);
        printf("\nLegal Baud Rates are : ");
        ct_disps(ct_br);          /* Display all values */
        exit();
        }
return retval;            /* Return code */
}

set_baud(pdevc,baudc)     /* Set the baud rate of the specified device */
int pdevc;                /* Physical device code */
short baudc;              /* Baud rate code */
                          /* On some systems this may have to be a
                             two-byte (unsigned) value  */
{
short *baud_rc;           /* Pointer to the baud rate constant */
                          /* On some systems this may have to be a
                             two-byte (unsigned) value  */
/* Note: the respective codes for accessing the baud rate constants
   via the get_cba (get configuration block address) function are:
        Device #0 = 19,  #1 = 21, #2 = 23. This function uses this
   mathematical relationship  */

        /* Set up pointer to the baud rate constant */
baud_rc = get_cba(CB_D0_BRC + (pdevc << 1));

        /* Then set the baud rate constant */
*baud_rc = baudc;

        /* Then call the BIOS initialization routine */
bios(CIOINIT,pdevc);
}

show_baud()               /* Show current baud rate */
{

int pdevn;                /* Physical device number */
short baudc;              /* Baud rate code */
                          /* On some systems this may have to be a
                             two-byte (unsigned) value  */
short *baud_rc;           /* Pointer to the baud rate constant */
                          /* On some systems this may have to be a
                             two-byte (unsigned) value  */
/* Note: the respective codes for accessing the baud rate constants
   via the get_cba (get configuration block address) function are:
        Device #0 = 19,  #1 = 21, #2 = 23. This function uses this
   mathematical relationship  */

printf("\nCurrent baud rate settings are :");


for (pdevn = 0; pdevn <= MAXPDEV; pdevn ++)     /* All physical devices */
        {
                /* Set up pointer to the baud rate constant --
                   the code for the get_cba function is computed
                   by adding the physical device number *2 to
                   the Baud Rate code for device #0 */

        baud_rc = get_cba(CB_D0_BRC + (pdevn << 1));

                /* Then set the baud rate constant */
        baudc = *baud_rc;

        printf("\n\t%s set to %s baud.",
                ct_strc(ct_pdev,pdevn), /* Get ptr. to device name */
                ct_strc(ct_br,baudc) ); /* Get ptr. to baud rate */
        }
}

chk_use(argc)             /* Check correct usage */
int argc;                 /* Argument count */
{
```

Figure 11-9. (Continued)

```
if (argc == 1)
        {
        printf("\nThe SPEED utility sets the baud rate speed for each physical device.");
        printf("\nUsage is :  SPEED physical-device baud rate, or");
        printf("\n            SPEED SHOW     (to show current settings)");
        printf("\n\nValid physical devices are: ");
        ct_disps(ct_pdev);
        printf("\nValid baud rates are: ");
        ct_disps(ct_br);
        exit();
        }
}
```

Figure 11-9. (Continued)

PROTOCOL — Set Serial Line Protocols

The PROTOCOL utility shown in Figure 11-10 is used to set the protocol for a specific serial device.

The drivers for each physical device can support several serial line protocols. The protocols are divided into two groups, depending on whether they apply to data output by or input to the computer.

Note that the output DTR and input RTS protocols can coexist with other protocols. The strategy is first to set the required character-based protocol and then to set the DTR/RTS protocol. There is an example of this in the following console dialog:

```
P3B>protocol<CR>
PROTOCOL Vn 1.0 02/17/83
PROTOCOL sets the physical device's serial protocols.
        PROTOCOL physical-device direction protocol {message-length}

Legal physical devices are :
                TERMINAL
                PRINTER
                MODEM

Legal direction/protocols are :
                Output DTR
                Output XON
                Output ETX
                Input RTS
                Input XON

        Message length can be specifed with Output ETX.


P3B>protocol show<CR>
PROTOCOL Vn 1.0 02/17/83
        Protocol for TERMINAL - None.
        Protocol for PRINTER - Output XON
        Protocol for MODEM - Input RTS


P3B>protocol m o e 128<CR>
PROTOCOL Vn 1.0 02/17/83
        Protocol for TERMINAL - None.
        Protocol for PRINTER - Output XON
```

```
                 Protocol for MODEM - Output ETX  Message Length 128 bytes.


         P3B>protocol m o d<CR>
         PROTOCOL Vn 1.0 02/17/83
                 Protocol for TERMINAL - None.
                 Protocol for PRINTER - Output XON
                 Protocol for MODEM - Output DTR Output ETX  Message Length
                   128 bytes.
```

```
#define VN "\nPROTOCOL Vn 1.0 02/17/83"
/* PROTOCOL -- This utility sets the serial port protocol for the
   specified physical device. Alternatively, it displays the
   current protocols for all of the serial devices. */

#include <LIBRARY.H>

        /* Code tables used to relate ASCII strings to code values */
struct _ct ct_iproto[3];        /* Code table for input protocols */
struct _ct ct_oproto[4];        /* Code table for output protocols */
struct _ct ct_dproto[7];        /* Code table for displaying protocols */
struct _ct ct_pdev[MAXPDEV + 2];/* Physical device table */
struct _ct ct_io[3];            /* Input, output */


        /* Parameters on the command line */
#define PDEV argv[1]    /* Physical device */
#define IO argv[2]      /* Input/output */
#define PROTO argv[3]   /* Protocol */
#define PROTOL argv[4]  /* Protocol message length */


main(argc,argv)
int argc;
char *argv[];
{
printf(VN);     /* Display sign-on message */
setup();        /* Set up code tables */
chk_use(argc);  /* Check correct usage */

        /* Check if request to show current settings */
if (usstrcmp("SHOW",argv[1]))
        {               /* No -- assume a set is required */
        set_proto(get_pdev(PDEV),       /* Physical device */
                        /* Input/output and protocol */
                get_proto(get_io(IO),PROTO),
                PROTOL);        /* Protocol message length */
        }
show_proto();

} /* end of program */

setup()                 /* Set up the code tables for this program */
{
        /* Initialize the physical device table */
ct_init(ct_pdev[0],0,PN_T);     /* Terminal */
ct_init(ct_pdev[1],1,PN_P);     /* Printer */
ct_init(ct_pdev[2],2,PN_M);     /* Modem */
ct_init(ct_pdev[3],CT_SNF,"*"); /* Terminator */

        /* Initialize the input/output table */
ct_init(ct_io[0],0,"INPUT");
ct_init(ct_io[1],1,"OUTPUT");
ct_init(ct_io[2],CT_SNF,"*");           /* Terminator */

        /* Initialize the output protocol table */
ct_init(ct_oproto[0],DT_ODTR,"DTR");
ct_init(ct_oproto[1],DT_OXON,"XON");
ct_init(ct_oproto[2],DT_OETX,"ETX");
```

Figure 11-10. PROTOCOL.C, a utility that sets the protocol governing input and output of a specified serial device

```
ct_init(ct_oproto[3],CT_SNF,"*");        /* Terminator */

        /* Initialize the input protocol table */
ct_init(ct_iproto[0],DT_IRTS,"RTS");
ct_init(ct_iproto[1],DT_IXON,"XON");
ct_init(ct_iproto[2],CT_SNF,"*");        /* Terminator */

        /* Initialize the display protocol */
ct_init(ct_dproto[0],DT_ODTR,"Output DTR");
ct_init(ct_dproto[1],DT_OXON,"Output XON");
ct_init(ct_dproto[2],DT_OETX,"Output ETX");
ct_init(ct_dproto[3],DT_IRTS,"Input RTS");
ct_init(ct_dproto[4],DT_IXON,"Input XON");
ct_init(ct_dproto[5],CT_SNF,"*");
}

unsigned
get_pdev(ppdev)           /* Get physical device */
/* This function returns the physical device code
   specified by the user in the command line. */
char *ppdev;              /* Pointer to character string */
{
unsigned retval;          /* Return value */

retval = ct_parc(ct_pdev,ppdev);/* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Physical Device '%s' is invalid or ambiguous.",
                ppdev);
        printf("\nLegal Physical Devices are : ");
        ct_disps(ct_pdev);      /* Display all values */
        exit();
        }
return retval;                  /* Return code */
}

unsigned
get_io(pio)               /* Get input/output parameter */
char *pio;                /* Pointer to character string */
{
unsigned retval;                /* Return value */

retval = ct_parc(ct_io,pio);    /* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Input/Output direction '%s' is invalid or ambiguous.",
                pio);
        printf("\nLegal values are : ");
        ct_disps(ct_io);        /* Display all values */
        exit();
        }
return retval;                  /* Return code */
}

unsigned

get_proto(output,pproto)
/* This function returns the protocol code for the
   protocol specified by the user in the command line. */
int output;               /* =1 for output, =0 for input */
char *pproto;             /* Pointer to character string */

{
unsigned retval;                /* Return value */

if (output)                     /* OUTPUT specified */
        {
                /* Get code for ASCII string */
        retval = ct_parc(ct_oproto,pproto);
        if (retval == CT_SNF)           /* If string not found */
                {
                printf("\n\007Output Protocol '%s' is invalid or ambiguous.",
        pproto);
                printf("\nLegal Output Protocols are : ");
                ct_disps(ct_oproto);    /* Display valid protocols */
                exit();
                }
```

Figure 11-10. (Continued)

```
        }
else                            /* INPUT specified */
        {
                /* Get code for ASCII string */
        retval = ct_parc(ct_iproto,pproto);
        if (retval == CT_SNF)           /* If string not found */
                {
                printf("\n\007Input Protocol '%s' is invalid or ambiguous.",
        pproto);
                printf("\nLegal Input Protocols are : ");
                ct_disps(ct_iproto);    /* Display valid protocols */
                exit();
                }
        }
return retval;                  /* Return code */
}


set_proto(pdevc,protoc,pplength)/* Set the protocol for physical device */
int pdevc;                      /* Physical device code */
unsigned protoc;                /* Protocol byte */
char *pplength;                 /* Pointer to protocol length */
{
struct _ppdt
{
char *pdt[16];          /* Array of 16 pointers to the device tables */
} ;
struct _ppdt *ppdt;             /* Pointer to the device table array */
struct _dt *dt;                 /* Pointer to a device table */

ppdt = get_cba(CB_DTA); /* Set pointer to array of pointers */
dt = ppdt -> pdt[pdevc];

if (!dt)                /* Check if pointer in array is valid */
        {
        printf("\nError -- Array of Device Table Addresses is not set for device #%d.",
                pdevc);
        exit();
        }

if (protoc & 0x8000)    /* Check if protocol byte to be set
                           directly or to be OR ed in */
        {               /* OR ed */
        dt -> dt_st1 != (protoc & 0x7F);
        }
else
        {               /* Set directly */
        dt -> dt_st1 = (protoc & 0x7F);
        }

if ((protoc & 0x7F) == DT_OETX) /* If ETX/ACK, check for message
                                   length */
        {
        if (isdigit(*pplength))         /* Check if length present */
                {
                        /* Convert length to binary and set device
                           table field. */
                dt -> dt_etxml = atoi(pplength);
                }
        }
}


show_proto()            /* Show the current protocol settings */
{
struct _ppdt
{
char *pdt[16];          /* Array of 16 pointers to the device tables */
} ;
struct _ppdt *ppdt;             /* Pointer to the device table array */
struct _dt *dt;                 /* Pointer to a device table */
int pdevc;                      /* Physical device code */
struct _ct *dproto;             /* Pointer to display protocols */

ppdt = get_cba(CB_DTA); /* Set pointer to array of pointers */

        /* For all physical devices */
```

Figure 11-10. (Continued)

```
for (pdevc = 0; pdevc <= MAXPDEV; pdevc++)
        {
                /* Set pointer to device table */
        dt = ppdt -> pdt[pdevc];

        if (dt) /* Check if pointer in array is valid */
                {
                printf("\n\tProtocol for %s - ",ct_strc(ct_pdev,pdevc));
                        /* Check if any protocols set */
                if (!(dt -> dt_st1 & ALLPROTO))
                        {
                        printf("None.");
                        continue;
                        }

                        /* Set pointer to display protocol table */
                dproto = ct_dproto;
                while (dproto -> _ct_code != CT_SNF)
                        {
                                /* Check if protocol bit set */
                        if (dproto -> _ct_code & dt -> dt_st1)
                                {       /* Display protocol */
                                printf("%s ",dproto -> _ct_sp);
                                }
                        ++dproto;       /* Move to next entry */
                        }
                        /* Check if ETX/ACK protocol and
                           message length to be displayed */
                 if (dt -> dt_st1 & DT_OETX)
                        printf(" Message length %d bytes.",
                                dt -> dt_etxml);
                 }
        }
}


chk_use(argc)           /* Check for correct usage */
int argc;               /* Argument count on commmand line */
{
if (argc == 1)
        {
        printf("\nPROTOCOL sets the physical device's serial protocols.");
        printf("\n\tPROTOCOL physical-device direction protocol {message-length}");
        printf("\n\nLegal physical devices are :");
        ct_disps(ct_pdev);
        printf("\nLegal direction/protocols are :");
        ct_disps(ct_dproto);
        printf("\n\tMessage length can be specifed with Output ETX.\n");
        exit();
        }
}
```

Figure 11-10. (Continued)

ASSIGN — Assign Physical to Logical Devices

The ASSIGN utility shown in Figure 11-11 sets the necessary bits in the physical input/output redirection bits in the BIOS. It assigns a logical device's input and output to physical devices. Input can only be derived from a single physical device, while output can be directed to multiple devices.

Here is an example console dialog showing ASSIGN in action:

```
P3B>assign<CR>
ASSIGN Vn 1.0 02/17/83
ASSIGN sets the Input/Output redirection.
        ASSIGN logical-device INPUT physical-device
        ASSIGN logical-device OUTPUT physical-dev1 {phy_dev2..}
        ASSIGN SHOW    (to show current assignments)
```

```
Legal logical devices are :
                CONSOLE
                AUXILIARY
                LIST

Legal physical devices are :
                TERMINAL
                PRINTER
                MODEM


P3B>assign show<CR>
ASSIGN Vn 1.0 02/17/83
Current Device Assignments are :
        CONSOLE INPUT is assigned to -  TERMINAL
        CONSOLE OUTPUT is assigned to -  TERMINAL
        AUXILIARY INPUT is assigned to -  MODEM
        AUXILIARY OUTPUT is assigned to -  MODEM
        LIST INPUT is assigned to -  PRINTER
        LIST OUTPUT is assigned to -  PRINTER

P3B>assign a o t m p<CR>
ASSIGN Vn 1.0 02/17/83
Current Device Assignments are :
        CONSOLE INPUT is assigned to -  TERMINAL
        CONSOLE OUTPUT is assigned to -  TERMINAL
        AUXILIARY INPUT is assigned to -  MODEM
        AUXILIARY OUTPUT is assigned to -  TERMINAL PRINTER MODEM
        LIST INPUT is assigned to -  PRINTER
        LIST OUTPUT is assigned to -  PRINTER
```

```
#define VN "\nASSIGN Vn 1.0 02/17/83"

#include <LIBRARY.H>

struct _ct ct_pdev[MAXPDEV + 2];        /* Physical device table */


        /* Names of logical devices */
#define LN_C    "CONSOLE"
#define LN_A    "AUXILIARY"
#define LN_L    "LIST"
struct _ct ct_ldev[4];          /* Logical device table */

struct _ct ct_io[3];            /* Input, output */

        /* Parameters on the command line */
#define LDEV argv[1]    /* Logical device */
#define IO argv[2]      /* Input/output */


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

printf(VN);     /* Display sign-on message */
setup();        /* Set up code tables */
chk_use(argc);  /* Check correct usage */

        /* Check if request to show current settings */
if (usstrcmp("SHOW",argv[1]))
        {               /* No, assume a set is required */
```

Figure 11-11. ASSIGN.C, a utility that assigns a logical device's input and output to two physical devices

```
                /* NOTE : the number of physical devices to
                   process is given by argc - 3 */
        set_assign(get_ldev(LDEV),get_io(IO),argc - 3,argv);
        }
show_assign();

}

setup()                 /* Set up the code tables for this program */
{
        /* Initialize the physical device table */
ct_init(ct_pdev[0],0,PN_T);     /* Terminal */
ct_init(ct_pdev[1],1,PN_P);     /* Printer */
ct_init(ct_pdev[2],2,PN_M);     /* Modem */
ct_init(ct_pdev[3],CT_SNF,"*"); /* Terminator */

        /* Initialize the logical device table */
ct_init(ct_ldev[0],0,LN_C);     /* Terminal */
ct_init(ct_ldev[1],1,LN_A);     /* Auxiliary */
ct_init(ct_ldev[2],2,LN_L);     /* List */
ct_init(ct_ldev[3],CT_SNF,"*"); /* Terminator */

        /* Initialize the input/output table */
ct_init(ct_io[0],0,"INPUT");
ct_init(ct_io[1],1,"OUTPUT");
ct_init(ct_io[2],CT_SNF,"*");           /* Terminator */

}

unsigned
get_ldev(pldev)         /* Get logical device */
/* This function returns the logical device code
   specified by the user in the command line. */
char *pldev;            /* Pointer to character string */
{
unsigned retval;                        /* Return value */
retval = ct_parc(ct_ldev,pldev);        /* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Logical device '%s' is invalid or ambiguous.",
                pldev);
        printf("\nLegal logical devices are : ");
        ct_disps(ct_ldev);      /* Display all values */
        exit();
        }
return retval;                  /* Return code */
}

unsigned
get_io(pio)             /* Get input/output parameter */
char *pio;              /* Pointer to character string */
{
unsigned retval;                /* Return value */

retval = ct_parc(ct_io,pio);    /* Get code for ASCII string */
if (retval == CT_SNF)           /* If string not found */
        {
        printf("\n\007Input/output direction '%s' is invalid or ambiguous.",
                pio);
        printf("\nLegal values are : ");
        ct_disps(ct_io);        /* Display all values */
        exit();
        }
return retval;                  /* Return code */
}

set_assign(ldevc,output,argc,argv)      /* Set assignment (I/O redirection) */
int ldevc;                      /* Logical device code */
int output;                     /* I/O redirection code */
int argc;                       /* count of arguments to process */
char *argv[];                   /* Replica of parameter to main function */
{
unsigned *redir;                /* Pointer to redirection word */
int pdevc;                      /* Physical device code */
unsigned rd_val;                /* Redirection value */

        /* Get the address of the I/O redirection word.
```

Figure 11-11. (Continued)

```
           This code assumes that get_cba code values
           are ordered:
                Device #0, input & output
                Device #1, input & output
                Device #2, input & putput

           The get_cba code is computed by multiplying the
           logical device code by 2 (that is, shift left 1)
           and added onto the code for Device #0, input
           Then the output variable (0 = input, 1 = output)
           is added on   */
redir = get_cba(CB_CI + (ldevc << 1) + output);

rd_val = 0;     /* Initialize redirection value */

        /* For output, assignment can be made to several physical
           devices, so this code may be executed several times  */
do
        {
                /* Get code for ASCII string */
                /* NOTE: the physical device parameters start
                   with parameter #3 (argv[3]). However argc
                   is a decreasing count of the number of physical
                   devices to be processed, Therefore, argc + 2
                   causes them to be processed in reverse order
                   (i.e. from right to left on the command line) */

        pdevc = ct_parc(ct_pdev,argv[argc + 2]);

        if (pdevc == CT_SNF)            /* If string not found */
                {
                printf("\n\007Physical device '%s' is invalid or ambiguous.",
                argv[argc + 2]);
                printf("\nLegal physical devices are : ");
                ct_disps(ct_pdev);      /* Display all values */
                exit();
                }
                /* Repeat this loop for as long as there are
                   more parameters (for output only) */
        else
                {
                /* Build new redirection value by OR ing in
                   a one-bit shifted left pdevc places. */
                rd_val |= (1 << pdevc);
                }
        } while (--argc && output);

*redir = rd_val;        /* Set the value into the config. block */
}

show_assign()                   /* Show current baud rate */
{
int rd_code;                    /* Redirection code for get_cba */
int ldevn;                      /* Logical device number */
int pdevn;                      /* Physical device number */
unsigned rd_val;                /* Redirection value */
unsigned *prd_val;              /* Pointer to the redirection value */

/* Note: the respective codes for accessing the redirection values
   via the get_cba (get configuration block address) function are:
        Device #0 console input -- 5
        Device #0 console putput -- 6
        Device #1 auxiliary input -- 7
        Device #1 auxiliary output -- 8
        Device #2 list input -- 9
        Device #2 list output -- 10

   This function uses this mathematical relationship  */

printf("\nCurrent device assignments are :");

        /* For all get_cba codes */
for (rd_code = CB_CI; rd_code <= CB_LO; rd_code++)
        {
                /* Set pointer to redirection value */
        prd_val = get_cba(rd_code);
                /* Get the input redirection value */
```

Figure 11-11. (Continued)

```
          rd_val = *prd_val;       /* This also performs byte reversal */

                  /* Display device name. The rd_code is converted to a
                     device number by subtracting the first code number
                     from it and dividing by 2 (shift right one place).
                     The input/output direction is derived from the
                     least significant bit of the rd_code. */

          printf("\n\t%s %s is assigned to - ",
                  ct_strc(ct_ldev,(rd_code - CB_CI) >> 1),
                  ct_strc(ct_io,((rd_code & 0x01) ^ 1)));

                  /* For all physical devices */
          for (pdevn = 0; pdevn < 16; pdevn++)
                  {
                          /* Check if current physical device is assigned
                             by AND ing with a 1-bit shifted left pdevn times */
                  if (rd_val & (1 << pdevn))       /* Is device active? */
                          {       /* Display physical device name */
                          printf(" %s",ct_strc(ct_pdev,pdevn) );
                          }
                  }

          }
}


chk_use(argc)             /* Check for correct usage */
int argc;                 /* Argument count on commmand line */
{
if (argc == 1)
        {
        printf("\nASSIGN sets the Input/Output redirection.");
        printf("\n\tASSIGN logical-device INPUT physical-device");
        printf("\n\tASSIGN logical-device OUTPUT physical-dev1 {phy_dev2..}");
        printf("\n\tASSIGN SHOW    (to show current assignments)");
        printf("\n\nLegal logical devices are :");
        ct_disps(ct_ldev);
        printf("\nLegal physical devices are :");
        ct_disps(ct_pdev);
        exit();
        }
}
```

Figure 11-11. (Continued)

DATE — Set the System Date

The DATE utility shown in Figure 11-12 sets the system date in the configuration block, along with a flag that indicates that the DATE utility has been used. Other utility programs can use this flag as a primitive test of whether the system date is current.

Here is an example console dialog:

```
P3B>date<CR>
DATE Vn 1.0 02/18/83
DATE sets the system date. Usage is :
        DATE mm/dd/yy
        DATE SHOW (to display current date)

P3B>date show<CR>
DATE Vn 1.0 02/18/83
        Current Date is 12/18/82

P3B>date 2/23/83<CR>
DATE Vn 1.0 02/18/83
        Current Date is 02/23/83
```

```
#define VN "\nDATE Vn 1.0 02/18/83"

/* This utility accepts the current date from the command tail,
   validates it, and set the internal system date in the BIOS.
   Alternatively, it can be requested just to display the current
   system date. */

#include <LIBRARY.H>

char *date;              /* Pointer to the date in the config. block */
char *date_flag;         /* Pointer to date-set flag */
int mm,dd,yy;            /* Variables to hold month, day, year */
int mcount;              /* Match count of numeric values entered */
int count;               /* Count used to add leading 0's to date */

main(argc,argv)
int argc;
char *argv[];
{
printf(VN);              /* Display sign-on message */
date = get_cba(CB_DATE);         /* Set pointer to date */
date_flag = get_cba(CB_DTFLAGS);/* Set pointer to date-set flag */

if (argc != 2)           /* Check if help requested (or needed) */
        show_use();      /* Display correct usage and exit */

if (usstrcmp("SHOW",argv[1]))    /* Check if not SHOW option */
        {
                /* Convert specified time into month, day, year */
        mcount = sscanf(argv[1],"%d/%d/%d",&mm,&dd,&yy);
        if (mcount != 3)                /* Input not numeric */
                show_use();     /* Display correct usage and exit */

                /* NOTE: The following validity checking is
                   simplistic, but could be expanded to accommodate
                   more context-sensitive checking: days in the month,
                   leap years, etc. */
        if (mm > 12 || mm < 1)  /* Check valid month, day, year */
                {
                printf("\nMonth = %d is illegal.",mm);
                show_use();     /* Display correct usage and exit */
                }
        if (dd > 31 || dd < 1)
                {
                printf("\nDay = %d is illegal.",dd);
                show_use();     /* Display correct usage and exit */
                }
        if (yy > 90 || yy < 83) /* <=== NOTE ! */
                {
                printf("\nYear = %d is illegal.",yy);
                show_use();     /* Display correct usage and exit */
                }

                /* Convert integers back into a formatted string */
        sprintf(date,"%2d/%2d/%2d",mm,dd,yy);
        date[8] = 0x0A;         /* Terminate with line feed */
        date[9] = '\0';         /* New string terminator */

                /* Change " 1/ 2/ 3" into "01/02/03" */
        for (count = 0; count < 7; count+=3)
                {
                if (date[count] == ' ')
                        date[count] = '0';
                }

                /* Turn flag on to indicate that user has set date */
        *date_flag |= DATE_SET;
        }
printf("\n\tCurrent Date is %s",date);
}

show_use()               /* Display correct usage and exit */
{
printf("\nDATE sets the system date. Usage is :");
printf("\n\tDATE mm/dd/yy");
printf("\n\tDATE SHOW (to display current date)\n");
exit();
}
```

Figure 11-12. DATE.C, a utility that makes the current date part of the system

TIME — Set the System Time

The TIME utility shown in Figure 11-13 sets the current system time. Like DATE, TIME sets a flag so that other utilities can test that the system time is likely to be current.

Here is an example console dialog:

```
P3B>time<CR>
TIME Vn 1.0 02/18/83
TIME sets the system time. Usage is :
        TIME hh{:mm{:ss}}
        TIME SHOW (to display current time)


P3B>time show<CR>
TIME Vn 1.0 02/18/83
        Current Time is 13:08:44


P3B>time 5:47<CR>
TIME Vn 1.0 02/18/83
        Current Time is 05:47:00
```

```
#define VN "\nTIME Vn 1.0 02/18/83"

/* This utility accepts the current time from the command tail,
   validates it, and sets the internal system time in the BIOS.
   Alternatively, it can just display the current system time. */

#include <LIBRARY.H>

char *time;             /* Pointer to the time in the config. block */
char *time_set;         /* Pointer to the time set flag */
int hh,mm,ss;           /* Variables to hold hours, minutes, seconds */
int mcount;             /* Match count of numeric values entered */
int count;              /* Count used to add leading zeros to time */

main(argc,argv)
int argc;
char *argv[];
{
printf(VN);             /* Display sign-on message */
time = get_cba(CB_TIMEA);      /* Set pointer to time */
time_flag = get_cba(CB_DTFLAGS);      /* Set pointer to the
                                         time-set flag */
hh = mm = ss = 0;       /* Initialize the time if seconds or
                           minutes are not specified */

if (argc != 2)          /* Check if help requested (or needed) */
        show_use();     /* Display correct usage and exit */

if (usstrcmp("SHOW",argv[1]))   /* Check if not SHOW option */
        {
                /* Convert time into hours, minutes, seconds */
        mcount = sscanf(argv[1],"%d:%d:%d",&hh,&mm,&ss);
        if (!mcount)            /* Input not numeric */
                show_use();     /* Display correct usage and exit */

        if (hh > 12)            /* Check valid hours, minutes, seconds */
                {
                printf("\n\007Hours = %d is illegal.",hh);
                show_use();     /* Display correct usage and exit */
                }
```

Figure 11-13. TIME.C, a utility that makes the current time part of the system

```
        if (mm > 59)
                {
                printf("\n\007Minutes = %d is illegal.",mm);
                show_use();     /* Display correct usage and exit */
                }
        if (ss > 59)
                {
                show_use();     /* Display correct usage and exit */
                printf("\n\007Seconds = %d is illegal.",ss);
                }

                /* Convert integers back into formatted string */
        sprintf(time,"%2d:%2d:%2d",hh,mm,ss);
        time[8] = 0x0A;         /* Terminate with line feed */
        time[9] = '\0';         /* New string terminator */

                /* Convert " 1: 2: 3" into "01:02:03" */
        for (count = 0; count < 7; count+=3)
                {
                if (time[count] == ' ')
                        time[count] = '0';
                }
                /* Turn bit on to indicate that the time has been set */
        *time_flag |= TIME_SET;
        }

printf("\n\tCurrent Time is %s",time);
}

show_use()              /* Display correct usage and exit */
{
printf("\nTIME sets the system time. Usage is :");
printf("\n\tTIME hh{:mm{:ss}}");
printf("\n\tTIME SHOW (to display current time)\n");
exit();
}
```

Figure 11-13. TIME.C, a utility that makes the current time part of the system (continued)

FUNKEY — Set the Function Keys

The FUNKEY utility shown in Figure 11-14 sets the character strings associated with specific function keys. In the specified character string, the character "<" is converted into a LINE FEED character. Here is an example console dialog:

```
P3B>funkey<CR>
FUNKEY sets a specific function key string.
        FUNKEY key-number "string to be programmed<"
                    (Note : '<' is changed to line feed.)
                    (        key-number is from 0 to 17.)
                    (        string can be up to 16 chars.)
        FUNKEY SHOW        (displays settings for all keys)


P3B>funkey show<CR>
FUNKEY Vn 1.0 02/18/83
        Key #0 = 'Function Key 1<'
        Key #1 = 'Function Key 2<'

P3B>funkey 0 "PIP B:=A:*.*[V]<"<CR>

P3B>funkey show<CR>
FUNKEY Vn 1.0 02/18/83
        Key #0 = 'PIP B:=A:*.*[V]<'
        Key #1 = 'Function Key 2<'
```

```
#define VN "\nFUNKEY Vn 1.0 02/18/83"

#include <LIBRARY.H>


int fnum;                         /* Function key number to be programmed */
char fstring[20];                 /* String for function key */
struct _fkt *pfk;                 /* Pointer to function key table */

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

if (argc == 1 || argc > 3)
        show_use();

pfk = get_cba(CB_FKT);  /* Set pointer to function key table */

if (usstrcmp("SHOW",argv[1]))
        {
        if (!isdigit(argv[1][0]))
                {
                printf("\n\007'%s' is an illegal function key.",
                        argv[1]);
                show_use();
                }

        fnum = atoi(argv[1]);   /* Convert function key number */

        if (fnum > FK_ENTRIES)
                {
                printf("\n\007Function key number %d too large.",fnum);
                show_use();
                }

        if (get_fs(fstring) > FK_LENGTH)
                {
                printf("\n\007Function key string is too long.");
                show_use();
                }


        pfk += fnum;    /* Update pointer to string */
                /* Copy string into function key table */

                /* Check if function key input present */
        if (!(pfk -> fk_input[0]))
                {
                printf("\n\007Error : Function Key #%d is not set up to be programmed.",fnum);
                show_use();
                }
        strcpy(pfk -> fk_output,fstring);
        }
else            /* SHOW function specified */
        {
        printf(VN);             /* Display sign-on message */
        show_fun();
        }
}

get_fs(string)          /* Get function string from command tail */
char string[];          /* Pointer to character string */
{
char *tail;             /* Pointer to command tail */
short tcount;           /* Count of TOTAL characters in command tail */
int slen;               /* String length */

tail = 0x80;            /* Command line is in memory at 0080H */
tcount = *tail++;       /* Set TOTAL count of characters in command tail */
slen = 0;               /* Initialize string length */

while(tcount--)         /* For all characters in the command tail */
        {
        if (*tail++ == '"')     /* Scan for first quotes */
                break;
```

Figure 11-14. FUNKEY.C, a utility that sets the character strings associated with specific function keys

```
        }
if (!tcount)            /* No quotes found */
        {
        printf("\n\007No leading quotes found.");
        show_use();
        }

++tcount;               /* Adjust tail count */
while(tcount--)         /* For all remaining characters in tail */
        {
        if (*tail == '"')
                {
                string[slen] = '\0';    /* Add terminator */
                break;          /* Exit from loop */
                }
        string[slen] = *tail++; /* Move char. from tail into string */

        if (string[slen] == '<')
                string[slen] = 0x0A;
        ++slen;
        }
if (!tcount)            /* No terminating quotes found */
        {
        printf("\n\007No trailing quotes found.");
        show_use();
        }
return slen;            /* Return string length */
}


show_fun()              /* Display settings for all function keys */
{
struct _fkt *pfkt;      /* Local pointer to function keys */
int count;              /* Count to access function keys */
char *lf;               /* Pointer to "<" character (LINE FEED) */

pfkt = get_cba(CB_FKT); /* Set pointer to function key table */
for (count = 0; count <= FK_ENTRIES; count++)
        {
        if (pfkt -> fk_input[0])        /* Key is programmed */
                {
                        /* Check if at physical end of table */
                if (pfkt -> fk_input == 0xFF)
                        break;  /* Yes -- break out of for loop */
                strcpy(fstring,pfkt -> fk_output);
                        /* Convert all 0x0A chars to "<" */
                while (lf = strscn(fstring,"\012"))
                        {
                        *lf = '<';
                        }

                printf("\n\tKey #%d = '%s'",count,fstring);
                }
        ++pfkt;         /* Move to next entry */
        }
}


show_use()
{
printf("\nFUNKEY sets a specific function key string.");
printf("\n\tFUNKEY key-number \042string to be programmed<\042 ");
printf("\n\t            (Note : '<' is changed to line feed.)");
printf("\n\t            (        key-number is from 0 to %d.)",
        FK_ENTRIES-1);
printf("\n\t            (        string can be up to %d chars.)",
        FK_LENGTH);
printf("\n\tFUNKEY SHOW        (displays settings for all keys)");
exit();
}
```

Figure 11-14. (Continued)

Other Utilities

Because of space limitations, not all of the possible utility programs for the BIOS features can be shown in this chapter. Others that would need to be developed in order to have a complete set are

PUBLIC/PRIVATE
: This pair of utilities would turn the public files flag on or off, making the files in user 0 available from other user numbers or not, respectively.

SETTERM
: This program would program the CONOUT escape table, setting the various escape sequences as required. It could also program the characters in the function key table that match with those emitted by the terminal currently in use.

SAVESYS
: This utility would save the current settings in the long term configuration block.

LOADSYS
: This would load the long term configuration block from a previously saved image.

DO
: This utility would copy the command tail into the multi-command buffer, changing "\" into LINE FEED, and then set the forced input pointer to the multi-command buffer. As a result, characters from the multi-command buffer would be fed into the console input stream as though they had been typed one command at a time.

SPARE
: This utility would work in conjunction with the hard-disk bad-sector management in your disk drivers. It would spare out bad sectors or tracks on the hard disk. This done, all subsequent references to the sectors or tracks would be redirected to a different part of the disk.