Writing C Applications Using Tapes [5]

This chapter describes the ways in which you may work with the tape subsystem with C programs.

Before you can access the tape subsystem for file processing, you must reserve the required number of tape drives for each device type needed. After you have reserved the tape drives, you may specify the tape volume in which the files to be processed are located.

After you have the volumes mounted and positioned, you can begin processing the tape files. When processing is complete, release the reserved tape drives.

There are two levels of access to the tape subsystem. The recommended and easiest to use is the C library level, using flexible file routines. The second level is to use system calls, which requires much greater detail than the C library level.

This chapter discusses accessing the tape subsystem with the following approaches:

- C flexible file I/O library routines
- System call I/O
- Tape information requests
- Tape positioning requests
- End-of-volume requests
- Tape control requests

Note: The ER90 format is not available on systems that run the UNICOS/mk operating system or that have GigaRing support.

5.1 C flexible file I/O library routines

The flexible file I/O (FFIO) routines provide another way to perform tape I/O with the ease of use of system calls. The FFIO routines automatically recognize tape devices and use the appropriate buffering.

The C library routines `ffopen`(3), `ffread`(3), `ffwrite`(3), `ffseek`(3), `ffbksp`(3), `ffclose`(3), and `ffwrite`(3) provide the capability to read and

write records to tape, rewind the tape and backspace records, and read and write tape marks.

For IBM compatible devices, the `ffread`(3) and `ffwrite`(3) routines provide an interface that is sensitive to block boundaries and that returns information on tape block boundaries on request. For ER90 devices, `ffread`(3) and `ffwrite`(3) provide a way to perform I/O by using either the byte-stream mode or block mode of the device. With the FFIO layer, a rewind operation can be performed simply with a call to `ffseek`(3). Tape marks can be written with `ffweof`(3) (`ffweof`(3) is not supported for ER90 devices in byte-stream mode), and tape marks can be read with `ffread`(3). A call to `ffwrite`(3) can write a tape block of a designated number of bytes on a tape. A call to `ffread`(3) can read up to one tape block from a tape. Explicit information about tape block boundaries and the ability to read and write partial tape blocks is available through the use of optional parameters on `ffread`(3) and `ffwrite`(3).

The `ffpos`(3) and `fffcntl`(3) routines provide the same complete set of capabilities as available from Fortran including additional positioning, access to information about the current tape, and end-of-volume processing. The `ffpos`(3) and `fffcntl`(3) routines are available on all systems. Some of the functionality available with `ffpos`(3) and `fffcntl`(3) on IBM compatible devices are not available on ER90 devices.

The FFIO tape layer may be used with either byte-stream mode or block mode of the ER90 devices. When you use byte-stream mode, EOV processing, user tape marks, and some positioning functionality are not available with the FFIO tape layer. When you use block mode and the FFIO tape layer, each record written must be the same size as specified on the `-b` option of the `tpmnt`(3) command. An exception to this rule is the last record written before a tape mark or the end-of-file.

Figure 39 shows a program, called `cexam.c`. This program demonstrates how these routines can be used. For more information on the C library routines, see the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080 or the *UNICOS/mk System Libraries Reference Manual*, Cray Research publication SR–2680. For detailed information about I/O, see the *Application Programmer's I/O Guide*, Cray Research publication SG–2168.

```
#include <fcntl.h>
#include <sys/types.h>
#include <foreign.h>
#include <errno.h>
main()
{
   int ffd;
   int i,j;
   int buf[2000];
   int ret;
   ffd = ffopen("mytape", O_RDWR);
   if (ffd<0){
      printf("open failed, error = %d\n",errno)
      exit(1);
   }

    /************** Write 10 records, a tape mark, and 10 more records to tape */
    for (j = 0; j < 2; j++){
      for (i = 0; i < 10; i++){
         ret = ffwrite(ffd, buf, 800);
         if (ret < 800){
         printf("ffwrite returned %d\n",ret);
         printf("error = %d\n",errno);
         }
      }

    /************** Write a tapemark */
      ret = ffweof(ffd);
      if (ret < 0)
         printf("ffweof failed, error = %d\n",errno);
   }
    /************** Rewind the tape */
    ret = ffseek(ffd,0,0);
    if (ret != 0)
      printf("ffseek failed, error = %d\n",errno);
```

```
   /************** Read the tape until the first tape mark is reached. */
    for (;;){
      ret = ffread(ffd, buf, 16000);
      if (ret < 0) {
         printf("ffread failed, error = %d\n",errno);
         break;
      }
      else if (ret == 0)
         break;
   /* Just read a tape mark */
      else
         printf("We read %d bytes\n",ret);
   }

    /************** Close the file */
    ffclose(ffd);
}
```

Figure 39. C library routine usage

Figure 40 shows how to execute `cexam.c`:

```
cc cexam.c
rsv CART 1
tpmnt -v ISCSL -l sl -p mytape -g CART -r in -n -T
assign -F tape mytape
./a.out
rls -a
```

Figure 40. Executing `cexam.c`

The `fffcntl`(3) routine provides the capability to detect tape end-of-volume, and to do special end-of-volume processing. An example of special end-of-volume processing using the FFIO routines follows.

For more information about end-of-volume processing, see Section 4.1.7, page 65. As described in this section, you must check for EOV after each `ffwrite`(3), `ffweof`(3), or `ffread`(3) when EOV processing is requested. For output data sets, check for EOV after each `fffcntl`(3) using `cmd FC_GETTP`

or `cmd FP_GETPOS`. For output data sets, you should also ensure that the library and system have flushed their buffers, and then test whether the tape is at EOV, before calling any of the following routines:

- `ffseek`(3)
- `ffpos`(3) (with `cmd FP_SETPOS`, `FP_SETTP`, `FP_SKIPF`, or `FP_BSEEK`)
- `ffclose`(3)
- `fffcntl`(3) (with `cmd FP_CLOSEV` or `FP_SETSP(off)`)

To flush the buffers, call `fffcntl`(3) using `cmd FC_GETTP` and with the structure field `ffc_synch` set to 1.

To execute `cexam2.c`, shown in Figure 41, enter:

```
cc cexam2.c
rsv TAPE 1
tpmnt -v VOL1:VOL2 -g TAPE -p mytape -r in -n -T
./a.out
rls -a
```

Figure 41. Executing `cexam2.c`

The `fffcntl`(3) and `ffpos`(3) routines, shown in Figure 42, are on all systems. EOV processing with `fffcntl`(3) is not available for ER90 devices.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/iosw.h>
#include <foreign.h>
#include <errno.h>

#define BUFSIZ 4000
#define ERREXIT(a, b) {printf("%s error = %d\n",a,b); exit(1); }

main()
{
        int ffd, fftmp;
        int i;
        long bufcnt;
        int buf[BUFSIZ];
        int ret, eov = 0;
        struct ffc_chktp_s checktp;
        struct ffc_gettp_s gettp;
        struct ffp_settp_s settp;
        long pa[40];
        struct ffsw stat;

        ffd = ffopens("mytape",O_RDWR,0,0,&stat,"tape");
        if (ffd < 0)
                ERREXIT("open failed ",stat.sw_error);
        /*
         * Initiate special end-of-volume processing
         */
        if (ffsetsp(ffd, &stat) < 0)
                ERREXIT("ffsetsp failed ",errno);
        /*
         * Write until we reach EOV
         */
        do {
                if (ffwrite(ffd,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("ffwrite failed ",errno);
```

```
        /*
         * We must check for EOV status after each write
         */
        if (fffcntl(ffd, FC_CHECKTP, &checktp, &stat) < 0)
                ERREXIT("CHECKTP failed ",stat.sw_error);
        if (checktp.stat == 0)
                eov = 1;        /* Have reached eov */
} while(!eov);

/* Determine how many blocks are buffered */
gettp.ffc_glen = 40;
gettp.ffc_synch = 0;
gettp.ffc_pa = pa;
if (fffcntl(ffd, FC_GETTP, &gettp, &stat) < 0)
        ERREXIT("GETTP failed ",stat.sw_error);
bufcnt = pa[10] + pa[11];        /* blocks in library buffer + system */
/*
 * Start special end-of-volume processing
 */
if (fffcntl(ffd, FC_STARTSP, 0, &stat) < 0)
        ERREXIT("STARTSP failed ",stat.sw_error);
/*
 * We will write the last 2 blocks on this volume and the
 * blocks that are buffered on the next volume.
 * Position backward 2 blocks.
 */
settp.ffp_nbs_p = FP_TPOS_BACK;
settp.ffp_nb = 2;
settp.ffp_nvs_p = 0;
settp.ffp_nv = 0;
settp.ffp_vi = 0;
if (ffpos(ffd, FP_SETTP, &settp, 0, &stat) < 0)
        ERREXIT("GETTP failed ",stat.sw_error);
/*
 * Read 2 blocks from tape + buffered blocks and store them
 * in a temporary file that is memory resident.
 */
if ((fftmp = ffopens("tmpfile",O_RDWR | O_CREAT,0,0,&stat,
                     "mr.scr.novfl")) < 0)
        ERREXIT("Error opening temporary file ",
                     stat.sw_error);
```

```
       for (i = 0; i < bufcnt+2; i++) {
                if (ffread(ffd,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("ffread failed ",errno);
                if (ffwrite(fftmp,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("ffwrite failed ",errno);
}
        /*
         * Position back 2 blocks.
         */
        settp.ffp_nbs_p = FP_TPOS_BACK;
        settp.ffp_nb = 2;
        settp.ffp_nvs_p = 0;
        settp.ffp_nv = 0;
        settp.ffp_vi = 0;

if (ffpos(ffd, FP_SETTP, &settp, 0, &stat) < 0)
                ERREXIT("SETTP failed ",stat.sw_error);
        for ( i = 0; i < 2; i ++) {     /* write 2 tape marks */
                if (ffweof(ffd)  < 0)
                        ERREXIT("ffweof failed ",errno);
        }
        /*
         * Close this volume and mount the next one in volume  identifier list
         */
        if (fffcntl(ffd, FC_CLOSEV, 0, 0, &stat) < 0)
                ERREXIT("Closev failed ",stat.sw_error);
        /*
         * End special processing.
         */
        if (fffcntl(ffd, FC_ENDSP, 0, 0, &stat) < 0)
                ERREXIT("Endsp failed ",stat.sw_error);
        /*
         * Disable special processing.
         */
        if (fffcntl(ffd, FC_SETSP, 0, 0, &stat) < 0)
                ERREXIT("Setsp failed ",stat.sw_error);
        /*
         * Write the data saved at eov. First rewind the temporary
         * file.
         */
```

```
       if (ffseek(fftmp, 0, 0) < 0)
                ERREXIT("Rewind of temporary file failed ",errno);
        for (i = 0; i < bufcnt+2; i++) {
                if (ffread(fftmp,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("Ffread failed ",errno);
                if (ffwrite(ffd,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("Ffwrite failed ",errno);
        }
        /*
         * Write 5 more blocks of data.
         */
        for (i = 0; i < 5; i++) {
                if (ffwrite(ffd,buf,BUFSIZ) != BUFSIZ)
                        ERREXIT("Ffwrite failed ",errno);

}
        /*
         * Close the tape file.
         */
        ffclose(ffd);
}
```

Figure 42. Using C library routines for EOV processing

5.2 System call I/O

Tape I/O at the system call level requires you to work with many details. You have a choice of synchronous or asynchronous I/O, and buffered or unbuffered I/O. You need to be concerned with buffer addresses, block size, number of bytes, and exception conditions. You need to know about specific hardware requirements of different Cray Research systems.

5.2.1 Cray Research systems

This section briefly describes system call level I/O concerns, and then describes in detail transparent I/O.

IBM compatible tape devices support blocked I/O. ER90 tape devices support blocked I/O and byte stream I/O.

For synchronous read and write requests, you must specify the buffer address and the number of bytes to read or write.

The block size for read and write operations restriction is based on a field size of 48 bits for IBM compatible devices. The CRAY J90 series have a maximum block size of 128 Kbytes except for the Small Computer System Interface (SCSI) I/O processor (IOP), which has maximum block size of 64 Kbytes minus 1 byte.

When a tape is read, the block size must be larger than or equal to the largest block size on the tape. The block size is specified with the `-b` option on the `tpmnt`(1) command or from the header label.

For ER90 devices, a *blocked file section* type consists of blocks of the size specified by the `-b` option of the `tpmnt`(1)command. Blocks within the file section, excluding the last block, must be the same length. The block size must be in the range of 80 through 1,199,832 bytes in 8-byte increments.

ER90 file sections within a tape can have different block lengths. You can change the block length for a file section from the value specified with the `tpmnt`(1) command by using the `TPC_SDBSZ ioctl`(2) request. The argument to the `ioctl`(2) request is the new block length, which cannot exceed the value specified with the `-b` option on the `tpmnt`(1) command. The block length can be changed only when the tape is positioned at the beginning of a file section.

You can use transparent I/O requests for reading and writing tape files. When you use transparent I/O, you do not need to be concerned with block size. Your program treats the data as a stream of bytes. In addition, transparent I/O allows you to specify either buffered or unbuffered I/O.

For ER90 devices, a *byte stream file* type is composed of blocks that are 1 byte in length. The ER90 device cannot access data that begins at an odd-byte memory address, therefore, byte stream data must be input and output to the device in even increments.

When using asynchronous I/O for transparent I/O or multilist I/O, you must acknowledge any exceptional conditions returned by the `reada`(2) and `writea`(2) system calls. When an exceptional condition occurs, the tape driver removes your I/O requests from the queue. When the driver receives additional I/O requests, it cannot determine if the requests were issued before or after an exceptional condition was returned; erroneous results may be generated. For example, an error status is returned in the `sw_error` field of the `iosw` structure for the `reada`(2) or `writea`(2) system call.

You may receive one of these exceptional conditions if you perform one of the following actions:

- You use asynchronous multilist I/O while processing tape marks.

  You must send an acknowledgment after each user tape mark is read.

- You use asynchronous multilist I/O while processing user end-of-volume.

  You must send an acknowledgment after receiving the ENOSPC status from the reada(2) or writea(2) system calls.

If you receive one of these exceptional conditions, but are able to continue processing, you must acknowledge receiving the condition by issuing a TPC_ACKERR ioctl(2) call as shown in the following example:

```
ioctl(fd, TPC_ACKERR, 0);
```

The fd option specifies the file descriptor.

All I/O requests received by the driver in the time between returning the exceptional condition and receiving the ioctl(2) acknowledgment are terminated with the error code ETPDACKERR. After the tape driver receives the acknowledgment, all I/O requests are processed normally.

5.2.2 Transparent I/O

If you are using transparent I/O, data is treated as a stream of bytes. To specify transparent I/O, open the tape file and issue read(2) or write(2) requests. If you issue a read(2) or write(2) request without specifying transparent I/O, the I/O is transparent by default. Transparent I/O can be either buffered or unbuffered.

5.2.2.1 Transparent buffered I/O

If transparent buffered I/O is requested, user data is temporarily stored in a system buffer. Transparent buffered I/O is the default I/O request type (do not include the -U option on the tpmnt(1) command).

To read a tape file with transparent buffered I/O, use the read(2) system call. The tape driver reads data blocks into a system buffer before copying data into a user buffer. The user may read any number of bytes. The tape driver copies the same number of bytes from the system buffer to the user buffer.

The following example shows you how to read 100 bytes, followed by another request to read the next 3 bytes from a tape file that has a maximum block size of 10,000 bytes, using transparent buffered I/O. This example can be used on all IOS systems.

1. Specify the block size as 10,000 bytes in the tpmnt(1) command:

   ```
   tpmnt -b 10000 -v SCRSL -f FILE
   ```

2. Specify the `open`(2) and `read`(2) statements in your C program:

```
filedes = open("file",O_RDONLY);
i = read(filedes, buf, 100);  /* read 100 bytes */
i = read(filedes, buf, 3);    /* read 3 bytes */
    . . .
```

To write a tape file with transparent buffered I/O, use the `write`(2) system call. The number of bytes requested to be written are copied into the system buffer. For IBM compatible devices, when the number of bytes of data accumulated in the system buffer is equal to the block size specified by the `-b` option of the `tpmnt`(1) command, the block of data is written to tape. For ER90 devices, when the buffer becomes full, the buffer is written to tape.

The following example shows you how to write a tape file that has a maximum block size of 10,000 bytes, using transparent buffered I/O. This example can be used on all IOS systems.

1. Specify the block size as 10,000 bytes in the `tpmnt`(1) command:

```
tpmnt -b 10000 -f FILE -v SCRSL -n
```

2. Specify the `write` statement in your C program:

```
filedes = open("FILE", O_RDWR);
write(filedes, buf, 20000);/*write 20000 bytes*/
        /* 2 blocks will be written to tape */
write(filedes, buf, 20);  /* write 20 bytes */
```

5.2.2.2 Transparent unbuffered I/O

To request unbuffered I/O, specify the `-U` option on the `tpmnt`(1) command. No system buffer will be used for user I/O. All I/O operations are done to and from your I/O buffer.

For ER90 byte stream requests, the byte count must be specified in even increments (excluding the last I/O) and be less than or equal to the device request limit, `CE_MAX_BLOCKS`.

For ER90 blocked requests, the byte count for reads must be greater than the maximum block size. In addition, each read transfers one block. For writes, the byte count must be a multiple of the block size, excluding the last I/O request.

To read a tape file with transparent unbuffered I/O, use the `read`(2) system call. A `read`(2) request transfers a tape block into your I/O buffer. For IBM compatible devices and ER90 blocked I/O requests, the number of bytes

specified in the `read`(2) request must be larger than or equal to the maximum block size specified by the `-b` option on the `tpmnt`(1) command, and it must be a multiple of 4096 bytes. When a `read`(2) completes with no error, a tape block is transferred into your I/O buffer, and the specified number of bytes is returned.

Figure 43 shows you how to read a tape file to an IBM compatible device using transparent unbuffered I/O:

1. Specify the block size as 10,000 bytes in the `tpmnt`(1) command:

```
rsv
tpmnt -v 123456 -l sl -P x -b 10000 -U -g CART
```

2. Specify the `read`(2) statement in your C program:

```
#include <fcntl.h>
main()
{
   char   buf[4096*3]; /* need 3 x 4096 bytes to hold 10000 bytes */
   int    fd;
   int    bytes;

   fd = open("x", O_RDONLY);
   bytes = read(fd, buf, 4096*3);
                            /* must request multiple of 4096 bytes */
}
```

Figure 43. Reading from an IBM compatible device (unbuffered I/O)

Figure 44 shows you how to read a tape file from an ER90 device using transparent unbuffered blocked I/O:

1. Specify the block size as 10,000 bytes in the `tpmnt`(1) command:

```
rsv CART
tpmnt -v 123456 -l sl -P x -b 10000 -B -U -g CART
```

2. Specify the `read`(2) statement in your C program:

```
#include <fcntl.h>
main()
{
   char   buf[4096*3];     /* 3 x 4096 bytes needed to hole 10000 bytes */
   int    fd;
   int    bytes;

   fd = open("x", O_RDONLY);
   bytes = read(fd, buf, 4096*3);}
```

Figure 44. Reading from an ER90 device (unbuffered blocked I/O)

Note: If a tape is accessed as blocked I/O as in the previous example, but is actually a byte stream file, `4096*3` bytes will be returned. An error will not be returned on the I/O request, even though the actual file type differs from the requested type.

Figure 45 shows you how to read a tape file from an ER90 device using transparent unbuffered byte stream I/O:

1. No block size is to be specified in the `tpmnt`(1) command:

   ```
   rsv ER90
   tpmnt -v 123456 -l sl -P x -U -g ER90
   ```

2. Specify the `read`(2) statement in your C program:

```
#include <fcntl.h>
 main()
 {
   char   buf[10000];
   int    fd;
   int    bytes;

   fd = open("x", O_RDONLY);
   bytes = read(fd, buf, 10000);}
```

Figure 45. Reading from an ER90 device (unbuffered byte stream I/O)

Note: If a tape is accessed as byte stream as in the previous example, but is actually a blocked tape, an error will be returned on the I/O request as the byte count is not a multiple of 4096 bytes.

To write a tape file with transparent unbuffered I/O, use the `write`(2) system call. For IBM compatible devices and ER90 blocked I/O, each `write`(2) request results in a block written from your user buffer to tape. When the `write`(2) returns with no error, the data in the user buffer is written to tape as a block. For ER90, blocked I/O requests must match the size specified with the `-b` option on the `tpmnt`(1) command. Each ER90 byte stream request writes the number of bytes requested.

Figure 46 shows you how to write a tape file to an IBM compatible device using transparent unbuffered I/O:

1. Specify the block size as 10,000 bytes in the `tpmnt`(1) command:

```
rsv CART 1
tpmnt -v ISCSL -l sl -P x -b 10000 -U -n -g CART
```

2. Specify the `write`(2) statement in your C program:

```
#include <fcntl.h>
main()
{
     char    buf[10000];  /* write buffer */
     int     fd;
     int     bytes;

     fd = open("x", O_WRONLY);
     bytes = write(fd, buf, 10000);  /* 10000-byte block */
     bytes = write(fd, buf, 500);    /* 500-byte block */
}
```

Figure 46. Writing to an IBM compatible device (unbuffered I/O)

Figure 47 shows you how to write a tape file to an ER90 device using transparent unbuffered byte stream I/O:

1. Specify the block size as 10,000 bytes in the `tpmnt`(1) command:

```
rsv ER90
tpmnt -v 123456 -l sl -P x -B -U -n -g ER90
```

2. Specify the `write`(2) statement in your C program:

```
#include <fcntl.h>
main()
{
     char    buf[10000];  /* write buffer */
     int     fd;
     int     bytes;

     fd = open("x", O_WRONLY);
     bytes = write(fd, buf, 10000);  /* 10000-byte block */
}
```

Figure 47. Writing to an ER90 device (unbuffered byte stream I/O)

5.3 Tape information requests

A C program can obtain tape subsystem information using system calls or tape daemon requests. This section discusses obtaining tape subsystem information from the tape information table, a tape daemon request, and several `ioctl`(2) requests.

5.3.1 Tape information table

The tape information table holds information about the tape system and is available to tape users. It is initialized by the tape driver when the system is started. When the tape daemon starts, it updates the table with information from its startup file.

The tape information table is defined in the `tapetab.h` file and included in a program by using the following preprocessor statement:

```
#include        <sys/tapetab.h>
```

The tape information table is defined so that it is not necessary to recompile if new fields are added to it in the future. It consists of a header with fixed length fields, followed by a variable length section. Figure 48 shows the format of the header:

```
typedef struct tapetab_struct {
     word  tape_tabsize;         /* size of table in bytes */
     word  tape_hdrsize;         /* size of tapetab header in bytes */
     word  tape_maxsize;         /* max size allocated to hold tapetab */
     word  tape_ios_model;       /* model E ios */
     word  tape_flag;            /* flags indicating various status */
     word  tape_dev_major;       /* major device number of tape devices */
     word  tape_dev_driver;      /* tape device driver name */
     word  tape_file_major;      /* user tape files major device number */
     word  tape_file_driver;     /* tape file driver name */
     word  tape_max_dev;         /* maximum number of tape devices */
     word  tape_conf_up;         /* maximum number of devices configured up */
     word  tape_max_per_dev;     /* max bytes for buffers per device */
     word  tape_max_bufs;        /* max buffers per device */
     word  tape_bmx_max_cmdlist; /* max cmds in a bmx cmdlist request */
}    tapetab;
```

Figure 48. Tape information table header

New fixed length fields may be added at the end of the header section. Offsets of variable fields are included in the fixed length fields.

Variable length fields follow the header with offsets defined in the header. Offsets are measured in words from the beginning of the table. These fields contain data, such as names. Fields of character strings must be null terminated. Variable length fields always start on word boundaries.

There are two types of variable length fields: single-item fields and a list of fields. Single-item fields, such as the daemon request pipe name, require a word in the header to hold its offset. A list of fields consists of a variable length list of offsets pointing to the corresponding field and requires two words in the header to hold the number of items in the list and the offset of the list.

The example, shown in Figure 49, accesses the tape information table, extracting the maximum number of tape drives:

```
#include   <sys/table.h>
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/tapetab.h>

main()
{
     word    tabsize;          /* size of table */
     tapetab_struct  *tblp;    /* pointer to tape information table */

     /*
      * get the size of tapetab.
      */

     if (tabread(TAPETAB, (char *)&tabsize, sizeof(word), 0)) {
       perror("can't read TAPETAB size");
       exit(1);
     }

     tblp = (tapetab_struct *)calloc(tabsize, 1);

     if (tabread(TAPETAB, (char *)tblp, tabsize)) {
       perror("can't read TAPETAB table");
       exit(1);
     }

     printf("max buffer size = %d bytes\n", tblp->tape_max_per_dev);
}
```

Figure 49. Using the tape information table

The `tabinfo`(2) and `tabread`(2) (see `tabinfo`(2)) system calls let you read a system table without reading `/dev/kmem`. The `tabinfo`(2) call describes table characteristics: location, header length, number of entries, and size of entry. Using the information returned by `tabinfo`(2), you can create a user buffer into which `tabread`(2) will read all or part of a table.

5.3.2 Tape daemon requests

The tape daemon request, called `TR_INFO`, lets you perform a tape status inquiry from within a C program. You must perform the following steps to send a tape daemon request and receive a reply.

1. Determine the request pipe name.

   The request pipe is automatically created for you when you issue the `rsv`(1)) command. Also, it is automatically deleted when you release all of your reservations by using the `rls`(1) command. The request pipe name must be the absolute path name, not just the file name portion. The directory of the request pipe is determined through the `#define USER_DIR` directive in file `tapedef.h`, which is set up at installation time to be the environment variable. The default is your environment variable `$TMPDIR`. For the file name portion, the `#define U_REQPIPE` directive in file `/usr/include/tapedef.h` defines the first part of the file name, which is appended by the job ID of your job. The default is `TAPE_REQ_`.

   The `#define MAXPATH` directive in file `tapedef.h` defines the longest path name minus one that the requests may use. If any path name is larger than `MAXPATH-1`, you must have the value of `MAXPATH` increased by your system administrator.

2. Build a reply pipe by using the `mknod`(2) system call. Open a pipe with an `open`(2) system call, keeping the pipe open until a reply returns.

   You can either build a reply pipe for each request and delete it after a reply has returned, or build a single reply pipe, using it for all of your requests. Regardless of the option you use, it is important to keep the reply pipe open until all replies have returned.

3. Place the reply pipe name in the request header. You must supply the absolute path name of the reply pipe.

4. Write the request into the pipe.

   Use the `write`(2) system call to write your request into the request pipe.

5. Read the reply header from the reply pipe.

   For each request submitted, the tape daemon sends a reply. Depending on the request you send, the reply may be just the reply header, or the reply header along with its data. To determine whether data has been returned, read the reply header from the reply pipe; if the size of the reply is larger than the reply pipe header, read in the rest of the reply.

You may use the echo field in the request and reply headers to help keep track of requests. The system copies what you input to the echo field of the request and reply headers.

Figure 50 shows a `TR_INFO` tape daemon request:

```
#include        <fcntl.h>
#include        <stdio.h>
#include        <tapedef.h>
#include        <tapereq.h>
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <sys/jtab.h>

extern  char    *calloc();
extern  char    *getenv();

main()

{
    char    *dirptr;
      char    *req_pipe_name;             /* request pipe file name */
      char    *rep_pipe_name;             /* reply pipe file name */

      struct  jtab    jobtab;             /* structure for job table info */
      struct  stat    status;             /* structure to stat tape file */

      int     req_fd;                     /* request pipe file descriptor */
      int     rep_fd;                     /* reply pipe file descriptor */
      int     tape_fd;                    /* tape file file descriptor */

       struct  trinfo  info_req;           /* tape info structure */
       struct  trinfor info_rep;           /* tape info reply structure */
       struct  rephdr  *rh;                /* reply header */

       int     c;
       int     size;                       /* total size of reply */

/*
 *      Check the status of the tape file (this assumes you have
 *      performed a tpmnt with -P or -p to a file "tapefile")
 *      and open the tape file
 */
  c = stat("tapefile",&status);
    if (c < 0) {
        perror("Stat failed for tapefile");
        exit(1);
 }
```

```
   tape_fd = open("tapefile", O_RDWR);

   if (tape_fd < 0) {
        perror("Unable to open tapefile");
        exit(1);
   }

/*
 *      Make the named reply pipe and open it
 *      Use tempnam() to get a unique temporary file name
 */

    rep_pipe_name = tempnam(NULL, NULL);
    c = mknod(rep_pipe_name, 010700);
    if (c < 0) {
        perror("Unable to mknod reply pipe");
        close(req_fd);
        close(tape_fd);
        exit(1);
   }

    rep_fd = open(rep_pipe_name, O_RDWR);
    if (rep_fd < 0) {
        perror("Unable to open reply pipe");
        close(req_fd);
        close(tape_fd);
        exit(1);
   }
/*
 *      Construct request pipe file name and open it
 */
  dirptr = calloc(1, MAXPATH);
    req_pipe_name = calloc(1, MAXPATH);

    dirptr = getenv(USER_DIR);
    c = getjtab(&jobtab);

    sprintf(req_pipe_name, "%s/%s%d",dirptr, U_REQPIPE, jobtab.j_jid);
    req_fd = open(req_pipe_name, O_WRONLY);
```

```
if (req_fd < 0) {
         perror("Unable to open request pipe");
         close(tape_fd);
         exit(1);
   }

  info_req.rh.size = sizeof(struct trinfo);
  info_req.rh.code = TR_INFO;
  info_req.rh.jid = jobtab.j_jid;
  info_req.st_dev = status.st_dev;
  info_req.st_ino = status.st_ino;
  strcpy(&(info_req.rh.rpn),rep_pipe_name);

  c = write(req_fd, &info_req, info_req.rh.size);
  if (c < 0) {
        perror("Unable to write to daemon's request pipe");
        close(req_fd);
        close(rep_fd);
        unlink(rep_pipe_name);
        close(tape_fd);
        exit(1);
  }

  close(req_fd);
  req_fd = 0;
/*
 *      Now read the reply back from the tape daemon from
 *      the reply pipe
 */
  rh = (struct rephdr *)calloc(1, sizeof(struct rephdr));

  c = read(rep_fd, (char *)rh, sizeof(struct rephdr));
  if (c < 0) {
        perror("Read of reply pipe failed");
        close(rep_fd);
        unlink(rep_pipe_name);
        close(tape_fd);
        exit(1);
  }
```

```
 size = rh->size;
  c = read(rep_fd, &info_rep, size);

  if (c < 0) {
        perror("Read of trinfor failed");
        close(rep_fd);
        unlink(rep_pipe_name);
        close(tape_fd);
        exit(1);
  }

/*
 *      Program can go on to print out selected fields of the tsdata
 *      structure returned, or use them for another purpose.
 */

   printf("ts_fcn (last function) = %o\n", info_rep.tsdata.ts_fcn);
   printf("ts_dst (device status) = %o\n", info_rep.tsdata.ts_dst);

/*
 *      Close remaining open files and clean up.
 */

   close(rep_fd);
   unlink(rep_pipe_name);
   close(tape_fd);
   exit(0);}
```

Figure 50. Using the TR_INFO request

Figure 51 shows the information that is returned from the #define TR_INFO directive:

```
struct tsdata {
  /*
   * Device status information
   */
int  ts_ord;                        /* Device ordinal */
int  ts_fcn;                        /* Last device function */
int  ts_dst;                        /* Last device status */
int  ts_dtr;                        /* Data transfer count */
int  ts_bmblk;                      /* Buffer memory block count */
int  ts_bmsec;                      /* Buffer memory sector count */
int  ts_pbmcnt;                     /* Partial block bytes in buffer memory */
int  ts_orsc;                       /* Outstanding sector count */
int  ts_orbc;                       /* Outstanding block count */
int  ts_bnum;                       /* Block number: Block number */
                                    /* relative to tape mark */
int  ts_utmnum;                     /* User Tape Mark number: */
                                    /* This only includes */
                                    /* tape marks embedded */
                                    /* in the user's data */
int  ts_tmdir;                      /* Direction from tape mark */
                                    /*    0 :   after tape mark */
                                    /*    1 : before tape mark */
/*
 * Tape file information
 */

char  ts_path[MAXPATH];             /* Path name */
char  ts_dgn[16];                   /* Device group name */
char  ts_dvn[16];                   /* Device name */
int   ts_year;                      /* Today's year */
int   ts_day;                       /* Today's day */
char  ts_fid[48];                   /* File id */
char  ts_rf[8];                     /* Record format */
int   ts_den;                       /* Density: */
                                    /*    1: 1600 bpi */
                                    /*    2: 6250 bpi */
int   ts_mbs;                       /* Max block size */
int   ts_rl;                        /* Record length */
int   ts_fst;                       /* File status: */
                                    /*    1  : new   */
                                    /*    2  : old   */
                                    /*    3  : append */
```

```
int   ts_lb;                    /* Label type: */
                                /* 1 : no label */
                                /* 2 : ANSII label */
                                /* 3 : IBM label  */
                                /* 4 : bypass label */
                                /* 5 : single tape mark label */
int  ts_fsec;                   /* File section number */
int  ts_fseq;                   /* File sequence number */
int  ts_ffseq;                  /* Fseq of 1st file on tape */
int  ts_ring;                   /* Write ring status:*/
                                /*    0 : ring out */
                                /*    1: ring in   */
int  ts_xyear;                  /* Expiration year */
int   ts_xday;                  /* Expiration day */
int  ts_first;                  /* First vsn of file */
char  ts_v1[80];                /* Vol1 label */
char  ts_h1[80];                /* Hdr1 label */
char  ts_h2[80];                /* Hdr2 label */
int   ts_numvsn;                /* Number of vsn */
int  ts_vsnoff;                 /* Offset to vsn list */
                                /* from beginning of */
                                /* struct tsdata */
int   ts_cvsn;                  /* Current vsn index */
int  ts_eov;                    /* User eov selected: */
                                /*    0 : eov processing off */
                                /*    1 : eov processing on  */
int  ts_eovproc;                /* If user is currently in */
                                /* special user EOV processing,*/
                                /* field is set to 1. Otherwise*/
                                /* field is 0. ts_eov would be 1*/
int  ts_urwtm;                  /* User read/write tape mark */
                                /*    0 : not requested   */
                                /*    1 : requested by -T */
                                /*    option of tpmnt command */
char ts_ba[8];                  /* block attribute */
int  ts_blank4;                 /* Unused */
int  ts_blank5;                 /* Unused */
```

```
/* Following the tsdata structure is the vsn list. It is
 * of variable length. The tsdata.ts_numvsn field is the number
 * of vsns in the list. The tsdata.ts_vsnoff field is the offset
 * (in bytes) to the beginning of the vsn list from the beginning of
 * the tsdata structure. The vsns are of the form char[8]. */
}
```

Figure 51. TR_INFO information

5.3.3 ioctl(2) requests

The ioctl(2) system call requests of TPC_EXTSTS and TPC_RDLOG ioctl(2) let you request information about the tape subsystem. The TPC_EXTSTS request lets you obtain information on ER90 devices and the TPC_RDLOG request can be used to obtain information on ER90 and IBM compatible devices.

5.3.3.1 ER90 TPC_EXTSTS request

To obtain the extended status of an ER90 device, use the TPC_EXTSTS ioctl request. The extended status consists of the following responses to device commands: report addressee status, attribute, operating mode, and report position.

The report addressee status response gives the state of the ER90 device (ready/not ready or online/offline), a description of the mounted volume, and the ER90 detailed status.

The attribute response returns the operational characteristics of the ER90 device (for example, the data block size, burst size, early end-of-media warning (EEW), location, and so on).

The operating mode response describes those attributes that were temporarily defined for the time the tape was positioned within the current partition.

The report position response contains the current absolute track address, the remaining partition capacity, and other tape location information (for example, at beginning-of-tape, past the EEW location, at a system zone, and so on).

Refer to the *ER90 Interface Control Document* provided by E-Systems, Inc., for a complete description of the command responses.

The extended status is obtained by issuing an ioctl(2) system call with a request code of TPC_EXTSTS, to either the tape path or to file TPDDEM_REQ. The tape path is the path specified on the tpmnt(1) command. TPDDEM_REQ is

a pseudo device used to issue requests to a device without users having to have the device assigned to them. If the request is issued to the pseudo device, the device name must be specified in the request. (`TPDDEM_REQ` is defined in the `tapedef.h` file.)

The argument of the `ioctl`(2) call must be a pointer to structure `ctl_extsts`. This structure is defined in Figure 52:

```
struct ctl_extsts   {
     int     device;
     char    *rep_addr;
     int     len_rep_addr;
     char    *attributes;
     int     len_attributes;
     char    *oper_mode;
     int     len_oper_mode;
     char    *report_pos;
     int     len_report_pos;
}
```

Figure 52. `ctl_extsts` structure

Set `rep_addr`, `attributes`, `oper_mode`, and `report_pos` to pointers to memory in which the response packets will be copied to receive responses to all of the commands. Set to `NULL` the memory pointers of the response packets that are not to receive only selected portions of the extended device status. Set the amount of memory allocated for the command in the `len_rep_addr`, `len_attributes`, `len_oper_mode`, or `len_report_pos` for each command requested. If the request is made to `TPDDEM_REQ`, `device` must be set to the device name. The length of each response packet is returned in the variables `len_rep_addr`, `len_attributes`, `len_oper_mode`, and `len_report_pos`.

The following restrictions apply to the ER90 `TPC_EXTSTS` request:

- The format or asynchronous I/O requests cannot be outstanding.
- Only the super user can issue this request through a pseudo device.
- The device must be configured up.

Note: If the operating mode response is requested and a cassette is not loaded, the cassette is blank, or the logical position has not been established, an operating mode response will not be returned.

Issuing requests to a device through the pseudo device suspends the current device activity until the extended status has been obtained.

Figure 53 shows how to obtain the extended status of an ER90 device by issuing a TPC_EXTSTS request using the tape path:

```
/*    Get the extended device status.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/sysmacros.h>
#include <sys/tpdctl.h>
#include <sys/epack.h>
#include <sys/epacki.h>
main()
{
       struct  ctl_extsts  ctl;
       char    rep_addr [MAX_IPI3_RESP_B];
       char    attributes [MAX_IPI3_RESP_B];
       char    oper_mode [MAX_IPI3_RESP_B];
       char    report_pos [MAX_IPI3_RESP_B];
       extern  int  errno;
       int         fd;
       int         c;
       /*
        *   Open the tape device path
        */
       fd = open( "tape_path", O_RDWR );
       if ( fd < 0 ) {
          perror( "Unable to open the device path" );
          exit(errno);
       }
       ctl.rep_addr = rep_addr;
       ctl.len_rep_addr = MAX_IPI3_RESP_B;
       ctl.attributes = attributes;
       ctl.len_attributes = MAX_IPI3_RESP_B;
       ctl.oper_mode = oper_mode;
       ctl.len_oper_mode = MAX_IPI3_RESP_B;
       ctl.report_pos = report_pos;
       ctl.len_report_pos = MAX_IPI3_RESP_B;
       /*
```

```
*   Issue the request for the extended device status.
        */
       c = ioctl( fd, TPC_EXTSTS, &ctl );
       if ( c < 0 ) {
          perror( "ioctl TPC_EXTSTS" );
          exit(errno);
       }
}
```

Figure 53. Using the ER90 TPC_EXTSTS request (tape path)

Figure 54 shows how to obtain the extended status on an ER90 device by issuing a TPC_EXTSTS request using a pseudo device:

```
/*
 *    Get the current position and remaining partition capacity of the
 *    mounted volume.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/sysmacros.h>
#include <sys/tpdctl.h>
#include <sys/epack.h>
#include <sys/epacki.h>
main()
{
       struct  ctl_extsts  ctl;
       char   report_pos[MAX_IPI3_RESP_B];
       extern  int  errno;
       int         fd;
       int         c;
       /*
        *   Open the pseudo device
        */
       ctl.device = 0;
       strncpy((char *)&ctl.device, "devname",strlen("devname"));
       fd = open( TPDDEM_REQ, O_RDWR );
       if ( fd < 0 ) {
          perror( "Unable to open the device path" );
          exit(errno);
       }
       bzero( (char *)&ctl, sizeof(struct ctl_abspos));
       ctl.len_report_pos = MAX_IPI3_RESP_B;
       ctl.report_pos = report_pos;
       /*
        *   Issue the request for the extended device status.
        */
       c = ioctl( fd, TPC_EXTSTS, &ctl );
       if ( c < 0 ) {
          perror( "ioctl TPC_EXTSTS" );
          exit(errno);
       }
}
```

Figure 54. Using the ER90 TPC_EXTSTS request (pseudo device)

5.3.3.2 ER90 read of the buffer log using TPC_RDLOG

The ER90 error log can be obtained by issuing an `ioctl`(2) system call, with a request code of `TPC_RDLOG`, to either the tape path or to file `TPDDEM_REQ`. The tape path is the path specified on the `tpmnt`(1) command. `TPDDEM_REQ` is a pseudo device used to issue requests to a device without the users having to have the device assigned to them. If the request is issued to the pseudo device, the device name must be specified in the request. (`TPDDEM_REQ` is defined in the `tapedef.h` file.)

The argument of the `ioctl`(2) call must be a pointer to structure `ctl_rdlog`. This structure is defined in Figure 55:

```
struct ctl_rdlog   {
     int     device;
     char    *device_log;
     int     length;
}
```

Figure 55. `ctl_rdlog` structure

The `device_log` field must be set to a pointer to the memory in which the ER90 error log will be copied. `length` must be set to the amount of memory allocated for the device log. If the request is made to `TPDDEM_REQ`, `device` must be set to the device name. The length of the device log will be returned in `length`.

The following restrictions apply to the `TPC_RDLOG` request:

- The format or asynchronous I/O requests cannot be outstanding.
- Only the super user can issue this request through a pseudo device.
- The device must be configured up.

  Note: Issuing requests to a device through the pseudo device suspends the current device activity until the extended status has been obtained.

Figure 56 shows how to read the ER90 error log by issuing a `TPC_RDLOG` request:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <sys/sysmacros.h>
#include <sys/tpdctl.h>
#include <sys/epack.h>
#include <sys/epacki.h>
#include <sys/er90_cmdpkt.h>

main()
{
       struct  ctl_rdlog  ctl;
       char    device_log[MAX_IPI3_RESP_B];
       int         fd;
       int         c;
       /*
        *   Open the tape device path
        */
       fd = open( "tape_path", O_RDWR );
/*
 *  or   ctl.device =0;
 *      strncpy((char *)&ctl.device, "devname", strlen("devname"));
 *      fd = open( TPDDEM_REQ, O_RDWR );
 */
       if ( fd < 0 ) {
         perror( "Unable to open the device path" );
         exit(1);
       }
       /*
        *   Issue the request for the ER90 Error Log.
        */
       ctl.length = MAX_IPI3_RESP_B;
       ctl.device_log = device_log;
       c = ioctl( fd, TPC_RDLOG, &ctl );
       if ( c < 0 ) {
         perror( "ioctl TPC_RDLOG" );
         exit(1);
       }
}
```

Figure 56. Using the ER90 TPC_RDLOG request

5.3.3.3 IBM compatible read of the buffer log using TPC_RDLOG

Note: The TPC_RDLOG request returns zeros on SCSI devices. It does not return an error code.

The IBM compatible buffer log can be obtained by issuing an `ioctl`(2) system call using the TPC_RDLOG request, as shown in Figure 57:

```
int   buflog[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };
int   i;

if ( ioctl( fd, TPC_RDLOG, buflog ) < 0) {
        perror( "Error reading buf log" );
        exit( 1 );
}

for ( i = 0 ; i < 8 ; i++ ) {
        printf( "0x%16.16x\n", buflog[i] );
}
```

Figure 57. Using the TPC_RDLOG request (IBM compatible)

The TPC_RDLOG request may be made after any read, write, or position request. It may be used to calculate compression ratio on the tape.

It is necessary to examine the sense information returned by the read buffer log request to determine compression ratios and distance from the end of the tape at any point while using the tape. The *IBM Hardware Reference Manual*, publication GA32-0127, provides detailed information on sense bytes and their formats.

Format 30 of sense bytes 32 through 43, provide counts for bytes processed by the channel and device. Channel counts reflect the number of bytes requested for the I/O operation, while the device counts reflect the number of bytes actually read or written by the device. The difference between the counts is the compression ratio achieved for the I/O operation.

Format 30 of sense byte 31 gives information about the length of the tape. It is possible to use this information in combination with the compression ratio information to determine approximately how much tape is used or remaining.

5.4 Tape positioning requests

C programmers use the `ffpos`(3) and `ffseek`(3) routines to implement tape positioning. For more information on positioning, see Section 5.1, page 81. Fortran programmers can position by using blocks and volumes as shown on Section 4.1.3, page 44 and Section 4.1.4, page 46.

5.5 End-of-volume requests

For information about user EOV processing from a Fortran program, see Section 4.1.7, page 65. Usually, volume switching is handled by the tape subsystem and is transparent to you. However, when user EOV processing is requested, you gain control at the end-of-tape and your program may perform special processing.

5.6 Tape control requests

You can use `ioctl`(2) system calls to control some characteristics of tapes. For ER90 devices, you can control the data block size and synchronize your program with the tape.

5.6.1 ER90 set data block size request

The data block size of a file section can be set by issuing an `ioctl`(2) system call `TPC_SDBSZ` request to a tape path. The tape path is the path specified on the `tpmnt`(1) command.

The argument of the `ioctl`(2) call is the data block size. The data block size must be in the range of 80 to 1,199,832 byes, and must be a multiple of eight.

The following restrictions apply to the `TPC_SDBSZ` request:

- The tape mount, format, or asynchronous request cannot be outstanding.
- The tape must be positioned at the beginning of a file section.
- The data block size cannot exceed the maximum block size specified by the `-b` option of the `tpmnt`(1) command.

Figure 58 shows how to set the data block size on an ER90 device:

```
#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <sys/tpdctl.h>

main()
{
       extern  int  errno;
       int     fd;
       int     c;
       /*
        *   Open the tape device path
        */
       fd = open( "tape_path", O_RDWR );
       if ( fd < 0 ) {
          perror( "Unable to open the tape path" );
          exit(errno);
       }
       /*
        *   Issue the request to set the DataBlock size.
        */
       c = ioctl( fd, TPC_SDBSZ, 32768 );
       if ( c < 0 ) {
          perror( "TPC_SDBSZ error" );
          exit(errno);
       }
}
```

Figure 58. Setting data block size

5.6.2 ER90 synchronize request

Synchronizing your program with the tape is accomplished by issuing an `ioctl`(2) system call `TPC_DMN_REQ` request to a tape path. The tape path is the path specified on the `tpmnt`(1) command.

The argument of the `ioctl`(2) call must be a pointer to structure `dmn_comm`. This structure is defined in Figure 59:

```
struct  dmn_comm {
     int     POS_REQ;
     int     POS_ABSADDR;
     int     POS_COUNT;
     int     POS_REP;
}
```

Figure 59. `dmn_comm` structure (synchronizing request)

There cannot be any outstanding asynchronous I/O requests for the `TPC_DMN_REQ` synchronize request to complete.

Note: If the previous request was a write request, data in the driver's buffer will be flushed to the tape. A synchronize request is then issued to the device, flushing the contents of the device's buffer to the tape. If the data in the system buffer is not a multiple of the data block size, a short block is output to the tape.

If the previous request was a read request and data is in the driver's buffer, the driver will backspace over the read ahead blocks. If there is a partial block in the buffer, the tape position is left after this block but the remainder of the block is deleted from the buffer.

Figure 60 shows how to synchronize your program with a tape on an ER90 device:

```
#include <sys/types.h>
#include <sys/fcntl.h>
#include <errno.h>
#include <tapereq>
#include <sys/tpdctl.h>

main()
{
       struct  dmn_comm  pos;
       extern  int  errno;
       int     fd;
       int     c;
       /*
        *   Open the tape device path
        */
       fd = open( "tape_path", O_RDWR );
       if ( fd < 0 ) {
          perror( "Unable to open the tape path" );
          exit(errno);
       }
       pos.POS_REQ  = TR_SYNC;
       /*
        *   Issue the sync request
        */
       c = ioctl( fd, TPC_DMN_REQ, &pos );
       if ( c < 0 ) {
          perror( "TPC_DMN_REQ error" );
          exit(errno);
       }
       /*
        *   Get the reply
        */
       c = ioctl( fd, TPC_DMN_REP, &pos );
       if ( c < 0 ) {
          perror( "TPC_DMN_REP failed" );
          exit(errno);
```

```
        }
        if ( pos.POS_REP ) {
           printf( "SYNC  error = %d", pos.POS_REP );
           exit(1);
        }
}
```

Figure 60. Synchronizing your program with a tape