Tape Troubleshooting [4]

Occasionally you may experience problems with the hardware or the software while running magnetic tapes. If so, there are certain steps you should take to try to clear the user, job, tape drive, or the tape daemon itself, and save the proper information to debug the problem.

This chapter describes the following troubleshooting topics:

- Tape drive or job problems
- Tape daemon problems
- `tpdfixup` utility
- Tracing
- Sample trace analysis
- `errpt`(8) utility
- Sample `errpt`(8) analysis
- `daemon.stderr` file
- `crash`(8) or`crashmk`(8) utility

4.1 Tape drive or job problems

If a tape drive appears to be hung, but the tape daemon is still responding to commands such as `tpstat`(1) and `tpgstat`(8), you can use the `tpfrls`(8) command to clear the user's tape reservation. You can determine the user ID and job ID to use with `tpfrls` from either `tpstat` or `tpgstat`. If this does not work, try the `tpclr`(8) command, with the tape device ID as an operand. If the problem appears to be hardware related, free the user by the preceding methods (check this with the `tpstat`(1) command). Then configure the drive down with the `tpconfig`(8) command, and discuss the problem with the appropriate hardware personnel.

4.2 Tape daemon problems

If the tape daemon (see `tpdaemon`(8)) is hung (no tapes moving, no response from any tape commands), and you must take the tape daemon down, first try the `tpdstop`(8) command. If this does not work, or the command hangs, determine the process ID of the tape daemon (by using the `ps`(1) command), and enter the following command line:

`kill -2` *pid*

The *pid* argument is the process ID of `tpdaemon`. If `kill -2` does not work, enter the following command line:

`kill -9` *pid*

If you want to report the preceding or other tape problems (such as the abnormal termination of the tape daemon), it is important that you save the trace files from the tape daemon. These files help to track down a tape problem. The trace files are kept in a directory set up during the initial installation of the tape daemon; see `TRACEPFX` in the `/usr/include/tapereq.h` file to find out where these files are kept.

The default installation of the trace files is in the `/usr/spool/tape` directory. Copy these files as follows:

`cd` *directory*
`cp /usr/spool/tape/trace.* .`

The *directory* argument is the directory in which you want to keep the trace files. It is also a good idea to create a file or note that explains what the problem was and specifies the devices that were affected: you may also want to keep a copy of the user job that seemed to cause the problem.

Another useful command is the `tpbmx`(8) command. `tpbmx` specified with the `-d` option displays the tape driver's tables for every device. It is recommended that you save a copy of the `tpbmx -d` output before attempting to execute `tpdstop` or attempting to terminate the daemon with the `kill`(1) command.

4.3 `tpdfixup` utility

The `tpdfixup` utility collects information pertinent to the online tape subsystem on Cray Research computer systems. A privileged user may run this script when a tape related problem occurs. The information is placed in a separate directory so it can be easily packaged and shipped for offline analysis. For the collected information to be of optimal use, tracing for the tape subsystem should be enabled. For detailed information, please contact the Cray Research Technical Support Center.

Before anything is copied to the information directory, the `tpdfixup` utility attempts to determine whether the tape daemon is in its normal state, and if not, runs a few checks for known hang situations.

The `tpdfixup` utility should be executed to gather information once trouble with the tape daemon is suspected prior to attempting to terminate the tape daemon.

4.4 Tracing

Tracing for the tape subsystem is turned on by default. All child processes created by the tape daemon have tracing enabled. While tracing is a very important tool for debugging tape subsystem problems, it uses additional CPU time. Tracing can be turned on and off by issuing the `tpset`(8) command. To turn tracing off, enter the following command:

```
tpset -T off
```

To turn tracing on, enter the following command:

```
tpset -T on
```

If the stability of the tape subsystem at a site has been established, tape tracing may be an unnecessary overhead. The CPU cycles saved by turning tracing off depends on the mix of jobs submitted, because some tape operations generate more trace information than others.

When tracing is turned off, the tape daemon and its child processes still trace entry to and exit from child processes and abnormal termination of tape processes. Abnormal terminations include those induced by the operator and terminations caused by errors within the tape subsystem. A tape mount request canceled by an operator or interrupted user job is considered an abnormal termination induced by the operator.

The option of turning tracing off for the tape subsystem allows sites running with a stable tape subsystem to substantially reduce the system and user time used by the tape daemon. This gain in system and user time must be weighed with the knowledge that some error information and all trace information will be lost in case of a tape daemon problem. The only way to analyze the problem is to turn tracing on, resubmit the job, and collect traces when the problem reappears.

4.5 Sample trace analysis

To obtain a complete picture of a problem, save trace information as soon as possible after you identify an error situation. You can use the tpdfixup utility to aid in the data gathering process.

This utility saves all the pertinent trace files in /usr/spool/tape as well as kernel traces through the issuance of crash(8) or crashmk(8) commands (in particular tpt and tps). If the tape daemon is not hung, the display command output is also saved. When you execute the utility, you are asked to comment on how the system was behaving at the time tpdfixup was run.

All of the trace files are circular. For instance, if a particular tape drive is hung, by the time it is noticed the tape daemon trace (trace.daemon) has probably been overwritten. However, the drive trace (trace.bmx###) and the kernel drive trace should provide some useful information. By default, the drive traces are 409600 bytes in length while the trace.daemon file is 10 times that value (the default is 4096000 bytes). This parameter is configurable in the tape configuration file.

Each time a tape daemon routine is entered, tracing for that routine begins. This is done by using the FUNC function defined in the tape.h file. RETURN and EXIT, also defined in tape.h, indicate when the routine is done.

Within each routine, you can place calls to the trace function to obtain more detailed information. By using this information, you can trace the paths that the software took to perform various tape functions.

When tpdaemon(8) forks off its children, (for example, opentdt and readerr) their trace information is written into the respective tape daemon device traces (trace.bmx###). There are also trace files for avrproc, stknet, esinet, and tcpnet. By using all of the appropriate traces, you can obtain the entire picture of what was happening when a failure occurred.

4.5.1 Trace information

The following example shows the information you obtain from a trace line.

```
10:59:58 151257598.1241 1450 tpdaemon mounttp function entered
^^^^^^^^ ^^^^^^^^^^^^^^ ^^^^ ^^^^^^^^ ^^^^^^^ ^^^^^^^^^^^^^^^^.......
AAAAAAAA BBBBBBBBBBBBBB CCCC DDDDDDDD EEEEEEE FFFFFFFFFFFFFFFF.......
```

The fields in this line are labeled as follows:

Field	Description
A	References the wall clock time. Having this time available is helpful in relating events in one trace to other traces, errpt(8) files, console messages or daemon.stderr messages.
B	References the real time clock. You use this time when timing issues are more important. It helps to determine whether the events truly took place in the proper order.
C	References the process number of the main routine. In the trace.daemon file, this value will invariably be tpdaemon(8); in the trace.bmx### files, the value will be the particular child tpdaemon(8) forks off to process the request (for example, opentdt, or writeerr).
D	Identifies the main routine.
E	References the particular routine called by the main routine. In this example, the routine is named mounttp.
F	Provides detailed trace information about the entry. This example shows that the mounttp function was entered.

4.5.2 Trace example

The following example shows what happens when a user issues an rsv(1) command. The listing contains fields E and F of the trace information from trace.daemon.

```
(Start of trace)
getreq function entered
getreq request came from /usr/spool/tape/daemon.request
getreq request X
getreq 000312fe  5472657148000470  0000000000000214  TreqH..p........
getreq 00031300  0000000000000293  0000000000000000  ................
getreq 00031302  0000000000000000  0000000000000000  ................
getreq 00031304  2f7573722f73706f  6f6c2f746170652f  /usr/spool/tape/
getreq 00031306  5c36353972737635  3439353700000000  659rsv54957....
getreq 00031308  0000000000000000  0000000000000000  ................
getreq                 ***** same *****
getreq 0003130e  0000000000005b6e  0000000000002e3c  ......[n.......<
getreq 00031310  0000000000005b6e  0000000000000000  ......[n........
getreq 00031312  0000000000000000  0000000000000000  ................
getreq 00031314  2f746d702f6a746d  702e303030363539  /tmp/jtmp.000659
getreq 00031316  612f544150455f52  45515f3635390000  a/TAPE_REQ_659..
```

```
getreq 00031318  0000000000000000  0000000000000000  ................
getreq                ***** same *****
getreq 0003131e  2f636c6f7564792f  75362f6261722f74  /cloudy/u6/bar/t
getreq 00031320  6170652e6d736700  0000000000000000  ape.msg.........
getreq 00031322  0000000000000000  0000000000000000  ................
getreq                ***** same *****
getreq 00031328  6261720000000000  0000000000000000  bar.............
getreq 0003132a  0000000000000001  4341525400000000  ........CART....
getreq 0003132c  0000000000000000  0000000000000001  ................
getreq 0003132e  0000000000000000  0000000000000000  ................
getreq                ***** same *****
getreq 00031342  0000000000000000  0000000000000063  ...............c
getreq 00031344  0000000000000063  0000000000000063  .......c.......c
getreq                ***** same *****
getreq 0003134a  0000000000000000  0000000000000000  ................
getreq 0003134c  0000000000002e3c  0000000000000009  .......<........
getreq 0003134e  00000000000003ef  0000000000000003  ................
getreq 00031350  0000000000003128  00000000000030f6  ......1(......0.
getreq 00031352  0000000000000000  0000000000000000  ................
getreq                ***** same *****
getreq 0003138a  0000000000000000  0000000000000000  ................
getreq getreq returns : code = 1
```

4.5.2.1 Source

The tape daemon checks its request pipes and determines a request is pending. The `getreq` function is entered as shown by the trace entry. While you examine the trace information, you may want to access the `tpdaemon`(8) source. Following the code in `getreq.c` is a trace entry:

```
if (reqhdr.code != TR_TPS) {                    /* don't dump tpstat */
        trace(func,"request came from %s",reqfsp->fn);
        DUMP("request",reqp,reqhdr.size);
}
```

This code traces from where the request came as well as dumping the request. If the request is a `tpstat`(1) command, it is not dumped because the `tpstat`(1) command is issued so often. To determine what the request is, examine the code in word one of the request. In this example, word 1 contains 0000000000000214. The information is dumped in hexadecimal as evidenced by the line `request X`. (A dump in octal would show `request O`.)

To identify the request, check the `tape.h` file:

```
fir013% grep 214 tape.h
#define TR_RSV          0x214                   /* reserve devices */
```

The request structures for each request are generally contained in the files named `tr` *xxxx.h*. *xxxx* refers to the command issued. To examine the request structure for this example, look in the `trsv.h` file. If a structure does not have its own header (`.h`) file, it is probably located in `tape.h`, the mount tape structure.

Within the `tpdaemon`(1) source is a series of `case` statements. Based on the request code, `tpdaemon`(1) calls the necessary function. In this instance, the request code of x214 corresponds to `TR_RSV`.

```
(from tpdaemon.c)
                case TR_RSV :
                        cfunc = rsvdev;
                        break;

(Trace continued)
rsvdev function entered
gettusr function entered
gettusr gettusr returns : code = 0
addq function entered
addq addq returns : code = 157881
dgpavail function entered
dgpavail dgpavail returns : code = 1
addrsv function entered
gettrsv function entered
gettrsv gettrsv returns : code = 201728
```

The `rsvdev` trace is the next function entered. It calls `gettusr` to determine if the user has already reserved a tape drive. `gettusr` returns a 0 indicating that no reserves are currently assigned to this user. Since a 0 is returned, the following `if` statement is false and the `if block` is bypassed.

```
(from rsvdev.c)
       if (tusrp = gettusr(reqp->rh.jid)) {    /* user found */
```

By looking at the code, you can deduce that this example was run on a system that did have security running because it does not contain any security trace entries.

Many of the `tpdaemon`(8) subroutines are contained within their own named `.c` file. Others are contained within various subroutines. If you cannot locate a particular routine, use a `grep`(1) command on the `tpdaemon`(1) source to find it.

`rsvdev` continues on. `addq` is then entered and returns the queue header pointer to `rsvdev`.

The `dgpavail` routine is called to determine if a device is available within the device group requested.

```
(from rsvdev.c)
        FUNC(dgpavail);

        for (i = 0; i < tdth.numdgp; i++) {
                trsvp = tdth.tusrqh.f->trsvp + i;
                if (!strcmp(trsvp->dgn,dgn)) {  /* found */
                        if (num > trsvp->num) {
                                rc = -1;
                        } else {
                                *trsvpa = trsvp;
                                rc = 1;
                        }
                        break;
                }
        }
        RETURN(rc);
```

The value that is returned, 1, indicates that a device is available. A particular return code is neither good nor bad based on its value; you must examine the source to determine the meaning of a code.

```
(from rsvdev.c)
                c = dgpavail(reqp->dgn[i].name,reqp->dgn[i].num,&tdtrsvp);
                if (c > 0) {                           /* available */
                        addrsv(tusrp,reqp->dgn[i].name,reqp->dgn[i].num);
```

Since `c` is greater than 0, the next block of code is executed. `addrsv` is called to add to tape reserved. `addrsv` calls `gettrsv` to return the address of the `trsv` structure. The code returned by `gettrsv` is the decimal address 201728, which converts to 31400 in hexadecimal. The `addrsv` trace dumps the `tusr` and `trsv` structures. The `trsv` structure is dumped from location x31400:

```
(Trace continued)
addrsv tusr X
addrsv 0003138f  0000000000000000  000000000002bcd4  ................
```

```
addrsv 00031391  2f746d702f6a746d  702e303030363539  /tmp/jtmp.000659
addrsv 00031393  612f544150455f52  45515f3635390000  a/TAPE_REQ_659..
addrsv 00031395  0000000000000000  0000000000000000  ................
addrsv              ***** same *****
addrsv 0003139d  000000000000000d  0000000000000293  ................
addrsv 0003139f  0000000000000000  0000000000000000  ................
addrsv 000313a1  0000000000000000  6261720000000000  ........bar.....
addrsv 000313a3  0000000000000000  0000000000005b6e  ..............[n
addrsv 000313a5  0000000000005b6e  2f636c6f7564792f  ......[n/cloudy/
addrsv 000313a7  75362f6261722f74  6170652e6d736700  u6/bar/tape.msg.
addrsv 000313a9  0000000000000000  0000000000000000  ................
addrsv              ***** same *****
addrsv 000313b1  0000000000000000  0000000000000001  ................
addrsv 000313b3  0000000000000000  0000000000000000  ................
addrsv              ***** same *****
addrsv 000313b7  0000000000031400  0000000000002e3c  ...............<
addrsv 000313b9  0000000000000000  0000000000000000  ................
addrsv              ***** same *****
addrsv 000313bd  0000000000002e3c  0000000000000009  .......<........
addrsv 000313bf  00000000000003ef  0000000000000003  ................
addrsv 000313c1  0000000000003128  00000000000030f6  ......1(......0.
addrsv 000313c3  0000000000000000  0000000000000000  ................
addrsv              ***** same *****
addrsv 000313fb  0000000000000000  0000000000000000  ................
addrsv trsv X
addrsv 00031400  4341525400000000  0000000000000000  CART............
addrsv 00031402  0000000000000001  0000000000000000  ................
addrsv 00031404  5441504500000000  0000000000000000  TAPE............
addrsv 00031406  0000000000000000  0000000000000000  ................
addrsv 00031408  5445535400000000  0000000000000000  TEST............
addrsv 0003140a  0000000000000000  0000000000000000  ................
addrsv 0003140c  3334393000000000  0000000000000000  3490............
addrsv 0003140e  0000000000000000  0000000000000000  ................
addrsv addrsv returns : code = 0
```

The next routine called, `bdmrsv`, sends an `ioctl`(2) system call about the reserve to the kernel.

```
(from rsvdev.c)
       if (ioctl(bmxfs.fd,BDM_RSV,jid) < 0) {
                errmsg(func,ETSYS,TM047,bmxfs.dvn,bmxfs.fn,"ioctl",
                                                        "BDM_RSV",errno);
                RETURN(errno);
```

```
        }
        usrmsg(func,TM000);                     /* tell user about it */

(Trace continued )
bdmrsv function entered
bdmrsv TM000 - tape resource reserved for you
bdmrsv bdmrsv returns : code = 0
```

4.5.2.2 Associated kernel trace entry

The kernel code to process the `ioctl`(2) system call is in `/usr/src/uts/c1/io/tpddem.c`. You can obtained this kernel information by issuing a `tpt tpdemreq` command from within the `tpdaemon`(8) command. These traces are in the oldest-to-latest order; the following is the latest or last trace entry:

```
tpddemct 0000000000000000002061 0000001000500000002006 .......1.. .....
```

`tpddemct` is entered as follows:

```
(from /usr/src/uts/c1/io/tpddem.c)
tpddemctl(vp, cmd, arg)
```

The trace is coded as:

```
(from /usr/src/uts/c1/io/tpddem.c)
        TPD_TRACE(io, ’tpddemct’, arg, UTPACK(cmd, vp));
```

From the `ioctl`(2) system call in `bdmrsv`, you can equate `vp` to `bmxfs.fd`, `BDM_RSV` to `cmd`, and `jid` to `arg`. Based on the kernel trace entry, 2061 should be the job ID. In this case, 1073 (decimal equivalent of 2061) is the job ID, and 10005 corresponds to the `BDM_RSV` command.

```
(from /usr/src/uts/c1/sys/tpddem.h)
#define TDM_RSV   010005      /* Mark job having device(s) reserved */
```

`rsvdev` then dumps `tusr` and `trsv`, calls `sendrep` to send the reply, and returns with a code of 0 that indicates successful completion.

```
rsvdev tusr X
rsvdev 0003138f  0000000000000000  000000000002bcd4  ................
rsvdev 00031391  2f746d702f6a746d  702e303030363539  /tmp/jtmp.000659
rsvdev 00031393  612f544150455f52  45515f3635390000  a/TAPE_REQ_659..
rsvdev 00031395  0000000000000000  0000000000000000  ................
rsvdev              ***** same *****
rsvdev 0003139d  000000000000000d  0000000000000293  ................
```

```
rsvdev 0003139f  0000000000000000  0000000000000000  ................
rsvdev 000313a1  00599e4eec1ee788  6261720000000000  .Y.N....bar.....
rsvdev 000313a3  0000000000000000  0000000000005b6e  ..............[n
rsvdev 000313a5  0000000000005b6e  2f636c6f7564792f  ......[n/cloudy/
rsvdev 000313a7  75362f6261722f74  6170652e6d736700  u6/bar/tape.msg.
rsvdev 000313a9  0000000000000000  0000000000000000  ................
rsvdev               ***** same *****
rsvdev 000313b1  0000000000000000  0000000000000001  ................
rsvdev 000313b3  0000000000000000  0000000000000000  ................
rsvdev               ***** same *****
rsvdev 000313b7  0000000000031400  0000000000002e3c  ...............<
rsvdev 000313b9  0000000000000000  0000000000000000  ................
rsvdev               ***** same *****
rsvdev 000313bd  0000000000002e3c  0000000000000009  .......<........
rsvdev 000313bf  00000000000003ef  0000000000000003  ................
rsvdev 000313c1  0000000000003128  00000000000030f6  ......1(......0.
rsvdev 000313c3  0000000000000000  0000000000000000  ................
rsvdev               ***** same *****
rsvdev 000313fb  0000000000000000  0000000000000000  ................
rsvdev trsv X
rsvdev 00031400  4341525400000000  0000000000000000  CART............
rsvdev 00031402  0000000000000001  0000000000000000  ................
rsvdev 00031404  5441504500000000  0000000000000000  TAPE............
rsvdev 00031406  0000000000000000  0000000000000000  ................
rsvdev 00031408  5445535400000000  0000000000000000  TEST............
rsvdev 0003140a  0000000000000000  0000000000000000  ................
rsvdev 0003140c  3334393000000000  0000000000000000  3490............
rsvdev 0003140e  0000000000000000  0000000000000000  ................
sendrep function entered
sendrep sendrep returns : code = 0
rsvdev rsvdev returns : code = 0
```

4.6 errpt(8) utility

The errpt(8) utility processes data collected by the error-logging mechanism (errdemon(8)) and generates a report of that data. The default report is a summary of all errors posted in the files specified on the command line. The options apply to all files. If you do not specify any files, errpt(8) attempts to use the /usr/adm/errfile file.

A summary report notes the options that can limit its completeness, records the time stamped on the earliest and latest errors encountered, and specifies the total number of errors of one or more error types. The number of times that errpt(8) has difficulty reading input data is included as read errors.

A detailed report contains, in addition to specific error information, all instances in which the error logging process was started and stopped, and the time changes (using the date(1) command) that may have occurred during the interval being processed. A summary of each error type included in the report is appended to a detailed report.

A report can be limited to certain records by the use of options.

For the tape subsystem, the errpt(8) command generates information useful for debugging both hardware and software. For more information, see the errpt(8) man page.

The following example will generate a detailed report about tape devices:

```
errpt -f -d tape
```

4.7 Sample errpt(8) analysis

The errpt(8) analysis available for SCSI protocols is more detailed than that for the block multiplexer (mux) and ESCON protocols. The samples in this section illustrate this difference.

4.7.1 Block mux and ESCON protocols

This analysis deals with errpt(8) tape errors for the block mux and ESCON protocols. Error information is generally logged in /usr/adm/errfile. When these logs are restarted, they are saved as files named errfile # where # is a sequential number starting with and incrementing. The errpt(8) program or the UNICOS olhpa(8) program reads the logs and formats the data. Error messages reported by errpt(8) are created by the bmxereclog routine called from the bmx routines in /usr/src/uts/c1/io.

You can also display these messages on the console by using the bmxconmsg routine. The console messages generally have the following form:

```
ebmx: cart04: unassign, command reject, C040002700000020 0000154400000000
AAAA  BBBBBB  CCCCCCCC  DDDDDDDDDDDDDD  EEEEEEEEEEEEEEEE EEEEEEEEEEEEEEEE
```

The fields in this line are labeled as follows:

Field	Description
A	Indicates the calling program, `ebmx`.
B	Indicates the device on which the error occurred.
C	Shows the `bmx` command that was issued.
D	Shows the resulting error message.
E	Usually records sense bytes 0 through 15. Verify by checking the specific error message in `ebmx.c`.

For aid in breaking down the sense bytes, see the appropriate IBM documentation.

The following command produced the sample `errpt`(8) record:

```
errpt -d tape -s 11011400 -e 11011500
```

The `-s` and `-e` parameters refer to the starting and ending times that were used. They are in the `mmddhhmm` format. The `-d` parameter indicates that `errpt`(8) should report on tape errors.

```
Tue Nov  1 14:53:53 1994
        Tape Error record Cluster 0 IOS 1 Device 150


        Volume:    Owner: 0   Command:

        (CART)  Error type: Drive assigned elsewhere  Final status: UNRECOVERED


        Initial channel: 036  Initial control unit: 013  Initial device: 000

        Final channel  : 000  Final control unit  : 000  Final device  : 000

        Request code: 0x9a  Response code: 0x0e

        Channel command: Assign

        Initial status (erpa): 0x045  Extended status: 0x2002

        Initial device status : 0x02  Final device status : 0x00

        Block: 0  Density: 0  Retry count: 00000

        Sense bytes: (hexadecimal)
```

```
00   -     41          40          80          45
04   -     00          00          00          20
08   -     01          40          33          e4
12   -     00          00          00          00
16   -     00          00          00          70
20   -     00          00          00          00
24   -     f6          80          34          72
28   -     20          50          00          00
```

This sample is a relatively straightforward `errpt`(8) record. If a tape job were involved, the volume, owner, and command fields would contain relevant information. However, the error type field indicates that the drive was assigned elsewhere with a final status of unrecovered.

The channel is octal 36, the control unit is octal 13, and the drive ID (initial device) is 0. You can verify this information in the tape configuration file:

```
{
        CONTROL_UNIT
                protocol = STREAMING ,
                status = UP ,
                path = ((036, 11))
        DEVICE
                name = 150 ,
                device_group_name = CART ,
                id = 00 ,
                type = 3480 ,
                status = DOWN ,
                loader = Operator
```

The request code of `x9a` indicates a command list, and the response code of `x0e` is a sequencer detected error. These commands are in the `/usr/include/sys/epackt.h` file under request codes to the IOS and IOS response codes.

```
/*
 * Define request codes to ios
 */

#define TCommandList                    0232

/*
 * IOS response codes
```

```
 */

#define RUnitCheck                        016
```

The channel command is `assign`. The ERPA code of `x045` can be located in the `/usr/include/sys/erec.h` file.

```
#define         T3480_DAE       0x45    /* Drive assigned elsewhere */
```

4.7.2 SCSI protocols

The sample shows the additional information that is available for SCSI protocols.

```
Tue Aug  6 15:48:53 1996
        Tape Error record Cluster 3 IOS 2 Device s9490s0


        Volume:    Owner: 40   Command:

        (CART)  Error type: Read data check  Final status: UNRECOVERED


        Initial channel: 002  Initial control unit: 002  Initial device: 000

        Final channel  : 000  Final control unit  : 000  Final device  : 000

        Request code: 0x9a  Response code: 0x0e

        Channel command: Load display

        Initial status (erpa): 0x023  Extended status: 0x400e

        Initial device status : 0x0e  Final device status : 0x00

        Block: 0  Density: 0  Retry count: 00000

        Sense bytes: (hexadecimal)
                 00   -     48           40           00           23
                 04   -     00           00           00           00
                 08   -     00           00           00           00
                 12   -     00           00           00           00
                 16   -     00           00           00           00
```

```
          20   -      00          00          00          00
          24   -      00          00          00          00
          28   -      00          00          00          00
          32   -      00          00          03          00
          36   -      00          00          00          00
          40   -      00          00          00          00
          44   -      11          01          00          00
          48   -      00          00          00          00
          52   -      00          00          00          00
          56   -      00          00          00          00
          60   -      00          00          00          00

SCSI Sense Byte 2 bits 3 - 0: 0x3(Medium Error)

SCSI Sense Byte 2 bits 7 - 5: 0x0

SCSI Sense Bytes 12/13: 0x1101(Read Retries Exhausted)

SCSI Sense bytes: (hexadecimal)
          00   -      00          00          03          00
          04   -      00          00          00          00
          08   -      00          00          00          00
          12   -      11          01          00          00
          16   -      00          00          00          00
          20   -      00          00          00          00
          24   -      00          00          00          00
          28   -      00          00          00          00
```

4.8 daemon.stderr file

The /usr/spool/tape/daemon.stderr file contains all tape daemon error messages. Therefore, this file contains debug information that helps diagnose errors. This file, along with the output from errpt(8), is useful for administrators when working on drive problems. It is also useful for debugging tape daemon problems when sent with other tape daemon trace files to Cray Research for offline analysis.

4.9 crash(8) or crashmk(8) utility

The crash(8) or crashmk(8) utility can help you discover and correct tape subsystem problems. This interactive utility can examine an operating system core image. It has facilities for interpreting and formatting the various control structures in the system and certain miscellaneous functions that are useful when examining a dump file.

The *core_filename* argument specifies where the system image can be found. The default value of *core_filename* is `/dev/mem`, which lets you use the `crash`(8) or `crashmk`(8) utility without an operand to examine an active system. If you specify the system image file, it is assumed to be a system core dump and the default process is set to that of the process active in the kernel at the time of the crash. This is determined by a value stored in a fixed location by the dump mechanism.

The following `crash`(8) or `crashmk`(8) commands are useful for tape problem solving:

`tpt` [*device1*][*device2*]...

Prints kernel level tape device traces. `tpt` called without any arguments prints out a table containing the device name (as seen in the `tpstat`(1) display); index (physical device name); and the start, middle, and end trace pointers for each device in the tape table. `tpt` called with a device name prints out traces for that device.

On UNICOS systems, `tpt` called with a dash (`-`) instead of *device1* dumps out traces for all tape devices in the system.

For more information concerning the `tpt` command on UNICOS/mk systems, use `help tpt` from within the `crashmk`(8) utility.

`tps` [*device1*][*device2*]...

(UNICOS systems only) Prints tape device structures. `tps` called without any arguments prints out tape I/O structures for all tape devices in the system. `tps` called with a device name prints out the tape structures associated with that device. `tps` called with a dash (`-`) instead of *device1* prints out tape structures for all tape devices in the system.