MPE/iX SCSI Pass-Through Programmer’s Guide

vCSY

Hewlett-Packard

Contents

Document Summary

The aim of this document is two fold:

1) To educate an application programmer on how to use ioctl() calls on SCSI pass through (SPT) device files to send SCSI commands directly to SCSI devices

2) Discuss the various recent enhancements in MPE/iX to make the SPT interface better and convenient to use

We begin with a discussion of SPT (what is it?, why use it?) followed by a program setup for using SPT to directly communicate with SCSI devices. We then discuss SYSGEN device configuration for making SPT calls on SCSI devices.

A description of the recent enhancements to SPT on MPE/iX follows next. The enhancements are:

- Facility in SCSI DMs to provide user notification of Bus Resets
- A user callable routine to convert a device's LDEV number to it's physical path.
- DUP capability on SPT device files to allow fork() calls in application programs
- Ability to use SPT device files across boots

A bug in the porting of the /sys/libIO.H header file has also been fixed.

Appendix-A includes a C program listing which shows how a SCSI Inquiry command can be executed on a SPT device file. The appendix also provides a snapshot of the results of executing the SCSI inquiry command on different types of devices.

Appendix-B has a listing of sample programs calling the new HPLDEVTOIOPATH user callable routine and also has a snapshot of the output.

Appendix-C has a list of JAG numbers for each of the above enhancements and the patch IDs in which they are available.

What is “SCSI Pass Through”?

“SCSI Pass Through” (SPT) refers to the ability of an application to send a SCSI Device Command through the MPE/iX OS and HP e3000 hardware to a SCSI Device: This is illustrated below in Figure #1.

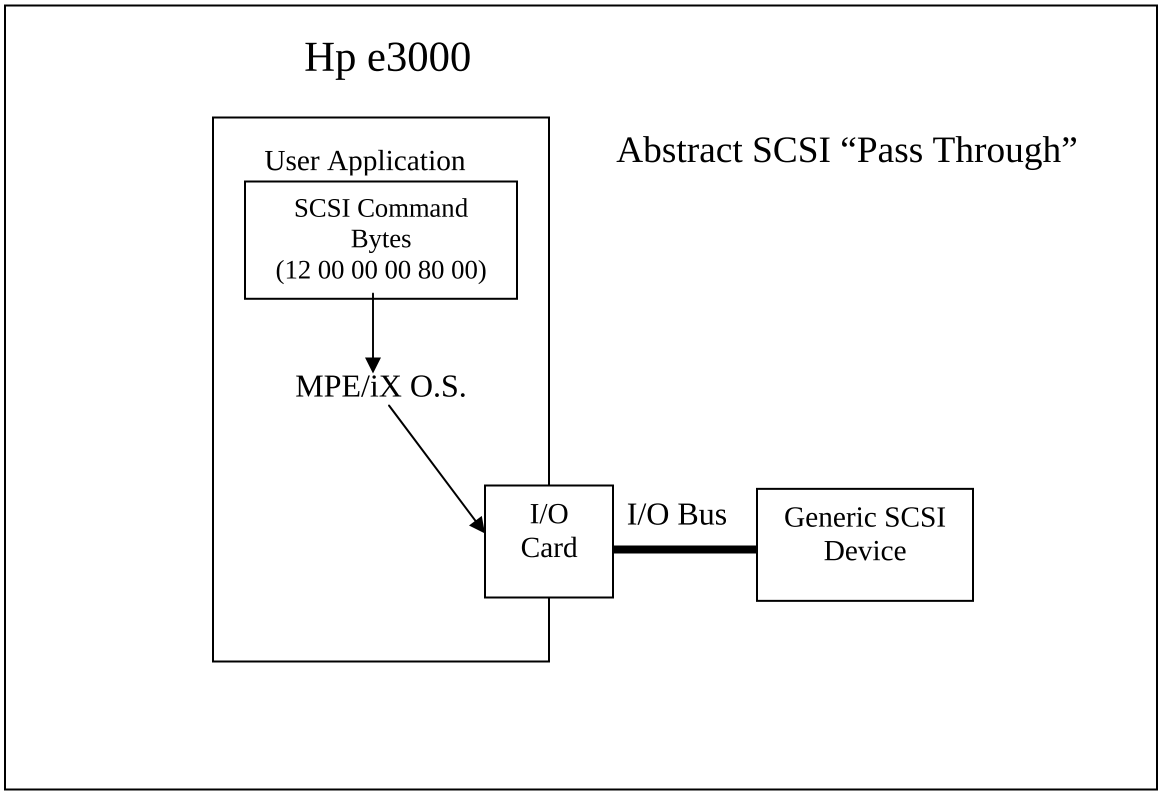

Figure #1

Why and when one might use SPT?

Most succinctly SPT allows a programmer to closely control the behavior of a SCSI device.

On MPE and most Operating Systems SCSI devices are controlled by a device-specific driver. Device-specific drivers, such as those for SCSI direct access (disk) and sequential access (tape) devices, coordinate device and driver states to accomplish correct logical device behavior. The SCSI pass-through interface enables one to interact directly with a SCSI device rather than depending upon MPE’s File System or “High-Level I/O” interfaces. With SPT MPE will allow both the usage of a “generic” SCSI device driver to allow one to interface to devices not supported by MPE/iX and the ability to talk to supported devices in a manner not currently allowed by MPE/iX Intrinsics.

The following are two examples of how this might be used.

1. Some SCSI hardware implementations include a controller that can monitor the status of devices in the enclosure and provide this information through SCSI commands. In practice, this information can be used to detect problem conditions like high temperatures or simply allow one to set an LED on a device to help pin-point it in a data center. SPT would enable programmers to obtain this information despite the fact that MPE/iX does not have a "SCSI Enclosure Controller Device Manager."

2. Some currently supported devices provide features that MPE/iX is not aware of. For instance, Ultrium Tapes allow a new type of File Indexing for faster searching of the media. MPE/iX has no knowledge of these features. A third party back-up tool might want to use SPT, instead of the MPE/iX File system, to be able to insert these media indexing marks on their tapes.

History and some caveats

The underpinning routines of this SCSI Pass-Through (SPT) interface have been present in MPE/iX since the 6.5 Release. In 6.5 the SPT interface was provided as an extension to MPE's POSIX interface for the exclusive use of the MPE/iX diagnostic tools (CSTM.DIAG.SYS). As these routines were originally designed as a porting enabler for an internal HP tool they may be a bit "rough around the edges". Therefore while this document describes a method of accessing these features of MPE, it may not describe every nuance and option of the interfaces. A working program and code examples are provided such that an experienced MPE/iX "C" programmer should be able to make use of what we have provided; this is not a document for beginners. Given the power of the SPT interface one should approach using it with the same caution, care, and thought given to MPE/iX Privilege Mode programs.

WARNING: Sending SPT commands to a device in use by MPE or other applications may result in data loss, data corruption and/or System Aborts. We do not recommend sending SPT commands to Disks with MPE/iX volumes present nor should one access tapes devices which are used for normal back-up or data logging purposes. Rule of thumb: Don't do SPT to tape or disk media you cannot recover, nor at a time you don't wish to cause a system outage.

We have tested, during this program or indirectly via the CSTM utilities, these interfaces with most of HP's Supported SCSI devices using 6, 10 and 12 Byte SCSI Commands. We have not tested with any "unsupported devices" nor with 20 Byte SCSI Commands. This interface is provided on an "as is" basis and no warrantee of usefulness or applicability should be assumed or is implied by the existence of this document.

Program Setup for SPT I/O

The programming example is derived from a program which was written in “C” to perform SCSI pass-through commands to a tape device connected to an HP-UX system.

Appendix-A includes a listing and sample output of a C program that issues the SCSI Inquiry command using the MPE/iX SPT interface. The appendix also has a snapshot of the output from the inquiry command on various device types.

Overview

The following is an outline of important steps performed by the example program:

1. Validate input parameters
2. Create device file via *mknod().*
3. Open device file via *open().*
4. Issue SCSI Inquiry (hex 12) via *ioctl()*
    a. Check command status
    b. Print returned information.
5. Interpreting results.

In addition to the SCSI INQUIRY command the program also sends an illegal command to provide an example of error handling.

Input parameter validation.

MPE/iX manages devices in a hierarchical manner by using a physical path for each device. This hierarchical ordering results in an “I/O tree” structure similar to the HP-UX ioscan output. Knowing a SCSI device’s physical path, one can create a device file and then issue SCSI commands directly to it via the MPE/iX SPT interface.

Following the HP-UX convention, SPT routines were originally defined to use the I/O Path as a “key”. The SPT enhancements to MPE/iX include also the ability to use the more common MPE/iX Logical Device or LDEV to look up an I/O Path for use with SPT.

Once the device file for a SCSI device is ready, it can be used to issue SCSI commands to it directly in application programs. The file needs to be opened using the *open()* call, SCSI commands are sent to the devices using *ioctl().* Please note that all HP-UX ioctl features are not supported.

SPT Device File Creation

The SPT device file is created by mknod(). Before calling *mknod()*, a unique file name should be assigned to the file. A clear convention is to name the devices using the actual logical path OR ldev number and place these files in a unique location. In this program the device file name is made up using the prefix "/dev/tm_diag" followed by the device path. As paths almost always include a forward slash "/" which has special meaning for the file system, all the "/" are first converted to "_" (under bar) to produce a more "usable" name. For example a path of "10/4/4.12.0" becomes "10_4_4.12.0" and so the device file will be named "/dev/tm_diag10_4_4.12.0. The device file is then created via the following call:

```
mknod (file_name, S_IFCHR, makedev (0, query_minor_num));
```

The constant S_IFCHR specifies a character device file type; The Major number is set to zero. Query_minor_num should contain the value returned by *io_query*; this is a 24bit version of the DM port number; see 'Deriving the mknod() "minor number" using io_query ' below for further discussion.

This mknod can be done against any path/ldev configured with one of the following SCSI DMs: *scsi_disc_dm, scsi_disk_and_and_array_dm, scsi_tape_dm, scsi_tape2_dm, mo_scsi_pthru_dm.* Device configuration is discussed in detail in later in this document in the section titled: "Device Configuration using SYSGEN" .

Deriving the mknod() "minor number" using io_query

As mentioned above, the physical path of a device must be known in order to supply information for the call to *mknod()*. An enhancement to MPE/iX provides a new user callable interface that returns the physical path of a device given its logical device number (LDEV). This routine is described in detail in the section of this document titled: "LDEV number to I/O path Conversion."

To call *mknod()*, the physical path (in the form of a character array) should first be verified by *io_str_to_hw_path()*. This routine scans the MPE/iX "io tree" structure for the path specified, checks that it exists and then encodes the path to an internal type. This encoded path value is referenced by the device_ptr which is next used by *io_search* to return a "token" for accessing the device path.

```
  (void) io_str_to_hw_path (path_ptr, device_ptr);
```

io_search is then called to get the "token" for our path using the device_ptr returned from *io_str_to_hw_path*:

```
  token = 0;   /*  start at "root" of io tree  */
  token = io_search ( token, S_IOTREE, 0, "hw_path", device_ptr);
```

Finally, given this token, we can get the information needed to actually create the device file, the "minor number", from *io_query*.

```
    (void) io_query (token, S_IOTREE, "minor_num", &query_minor_num);
```

Opening the SPT device file

Open() is used to open the SPT device file after it has been created via *mknod()*. The file is opened as an ordinary disc file. The device file is actually a “sharable” file, so it is possible to have multiple opens from multiple processes. The application program could also create child processes via *fork()* and can issue many parallel SCSI commands to the device that is opened. But, it should be noted that the capability to use *fork()* on SPT device files is a recent enhancement to MPE/iX and requires the corresponding patch to be installed. For details on this enhancement, please refer the section “Recent Enhancements”.

The device file is closed using *close()* after use. *Close()* is done via the filenum returned by *open()*. The device file can be purged using the *unlink* call.

Issuing SCSI commands

SPT I/O is achieved by using *ioctl()* calls. Making the *ioctl()* call is very simple:

```
r = ioctl(fd, SIOC_IO, &command);
```

SIOC_IO specifies a SCSI Pass through request. The “&command” passes the address of the variable (a structure) which contains the SCSI command (called as the Command Data Bytes – CDB) and related information:

```
struct sctl_io command;
```

The sctl_io structure is defined in the “sys/scsi.h” file. This definition is exactly as it appears in the HP-UX “sys/scsi.h” file. There is a data_xfer field which is returned by the low level I/O layers to tell the actual number of bytes transferred. Typically, the initialization resembles:

```
memset(command, 0, sizeof(struct sctl_io));
command.flags = SCTL_READ;
command.cdb_length = 6;
command.cdb[0] = CMDtest_unit_ready;
command.cdb[4] = sizeof(struct inquiry_2);
command.data = buf;
command.data_length = 64*1024;
command.max_msecs = timeout;
```

sctl_io flags and commands supported are defined in sys/scsi.h. *sctl_io.data* is the buffer where the command output is returned. A detailed description of *ioctl* and the data structures involved is beyond the scope of this document. Please click here for a more detailed explanation. All SCSI commands are described as part of the SCSI Standards documents available at http:www.t10.org.

These *ioctl()* calls are “blocking” and follow a single-threaded path. i*octl()* does not return until the I/O is completed. Completion includes normal I/O completion as well as requests which are aborted due to timeouts or errors.

MPE/iX does security and bounds checking of the data and sense buffers. Again, there is no cross checking of the CDB vs. the other command fields; it is up the caller to be able to construct a valid CDB as well as make it consistent with the other parameters.

Please note most all *ioctl* functions that are supported on HP-UX are **not** supported in MPE/iX. Only SIOC_IO should be attempted; SIOC_EXCLUSIVE, SIOC_RESET_BUS are not implemented. Additional SCSI.H *ioctl* features like SIOC_IDENTIFY, SIOC_SYNC_CACHE etc. and functions like SIOC_SET_BUS_LIMITS etc., to control LUN, bus settings, are also not supported.

Interpreting results

A return value of 0 (zero) from *ioctl()* indicates a successful completion of the *ioctl* call. This does not however, indicate the actual result of the SCSI I/O command. The SCSI command that was passed to *ioctl* includes status values that indicate the status of the SCSI request. These status values need to be checked to ensure successful completion of the I/O request.

command.cdb_status and *command.sense_status* provide status information of the completed I/O request. If *command.cdb_*status does not equal the constant *S_Good* , it is indicative of an I/O error. If *cdb_status* is set to *S_Check_Condition*, then c*ommand.sense_status* provides more information about the error. The following “C” code fragment shows a simple example of checking these items:

```
if ( command.cdb_status != S_GOOD )
{
  printf("SCSI command returned status 0x%02x\n", command.sense_status);
  if ( command.cdb_status == S_CHECK_CONDITION )
  {
   if ( command.sense_status == S_GOOD )
   printf("Request sense returned status 0x%02x\n", command.sense_status);
   else
   {
       memcpy(&sense_data,&(command.sense),command.sense_xfer);
      printf("Sense error code: 0x%02x\n", sense_data.error_code);
      printf("Sense key: 0x%02x\n", sense_data.key);
      printf("Sense code: 0x%02x\n", sense_data.code);
      printf("Sense qualifier: 0x%02x\n", sense_data.qualifier);
   }
  }
}
else printf("Good status, %d bytes returned (in hex): \n",
             command.data_xfer);
```

The following are the SCSI error codes that can be returned:

```
S_GOOD                  0x00
S_CHECK_CONDITION       0x02
S_CONDITION_MET         0x04
S_BUSY                  0x08
S_INTERMEDIATE          0x10
```

```
S_RESV_CONFLICT         0x18
S_COMMAND_TERMINATED    0x22
S_QUEUE_FULL            0x28
S_I_CONDITION_MET       (S_INTERMEDIATE + S_CONDITION_MET)
```

The following are the sense keys:

```
S_NO_SENSE              0x00
S_RECOVERED_ERROR       0x01
S_NOT_READY             0x02
S_MEDIUM_ERROR          0x03
S_HARDWARE_ERROR        0x04
S_ILLEGAL_REQUEST       0x05
S_UNIT_ATTENTION        0x06
S_DATA_PROTECT          0x07
S_BLANK_CHECK           0x08
S_VENDOR_SPECIFIC       0x09
S_COPY_ABORTED          0x0a
S_ABORTED_COMMAND       0x0b
S_EQUAL                 0x0c
S_VOLUME_OVERFLOW       0x0d
S_MISCOMPARE            0x0e
S_RESERVED              0x0f
```

Please see http://docs.hp.com/en/B3921-90008/B3921-90008.pdf for more details on using *ioctl* and interpreting results; additional information also at http://www.t10.org/.

Device Configuration using SYSGEN

WARNING: Sending SPT commands to a device in use by MPE or other applications may result in data loss, data corruption and/or System Aborts. We do not recommend sending SPT commands to Disks with MPE/iX Volumes present nor should one access tapes devices which are used for normal back-up or data logging purposes. Rule of thumb: Don’t do SPT to tape or disk media you cannot recover at a time you don’t wish to cause a system outage.

When a device is added using SYSGEN/IOCONFIG, based on the device ID supplied, a physical manager (pmgr, a.k.a Device Manager or DM) is assigned to the device. The DM used by a particular device can be seen when a ‘lp’ (list path) of the device’s physical path is done. The following DMs (pmgrs) in MPE/iX support SPT calls:
scsi_disc_dm (discs),
scsi_disk_and_and_array_dm (disc arrays),
scsi_tape_dm (DDS tape drives),
scsi_tape2_dm (DLT tape drives),
mo_scsi_pthru_dm (MO devices).

It is recommended that most users of SPT consider configuring SCSI devices with device ID=PTHRU which results in using the *mo_scsi_pthru_dm* pmgr. This device driver provides basic connectivity to SCSI devices without a lot of overhead.

We can also use certain generic IDs to configure devices. For instance, a disc can be configured with ID=LVDDISK, disc arrays with ID=HPDARRAY and a tape with ID=DDS. The appropriate pmgr is set based on the ID. This will still allow MPE/iX to use these devices. But, please be cautioned that SPT access to such devices may corrupt the MPE/iX functionality, meaning that MPE/iX may not be able to read/write to the device.

One may also use more common HP product IDs (e.g HPC1537A – a DDS tape drive) for HP devices As SCSI commands can also be issued to these devices via the SPT interface outlined in the previous section. Again, these devices will look just like other MPE/iX Devices and may therefore cause one to accidentally corrupt data.

For SPT access to devices that are not supported by MPE/iX, it is recommended to use device ID=PTHRU which results in using the *mo_scsi_pthru_dm* pmgr.

The following is a sample configuration of different device types:

```
LDEV:      1  DEVNAME: 00000001           OUTDEV:        0   MODE:
   ID: HPDARRAY                            RSIZE:       128   DEVTYPE: DISC
 PATH: 0/0/1/1.15.0                        MPETYPE:       4   MPESUBTYPE:  2
CLASS: DISC     SPOOL

LDEV:      7  DEVNAME: 00000007           OUTDEV:        0   MODE:
   ID: DDS                                 RSIZE:       128   DEVTYPE: TAPE
 PATH: 0/0/2/0.6.0                         MPETYPE:      24   MPESUBTYPE:  7
CLASS: TAPE     TAPE2    DDUMP    TAPE1
```

```
LDEV:     8  DEVNAME:                    OUTDEV:        0   MODE:
   ID: HPC1504B                           RSIZE:       128   DEVTYPE: TAPE
 PATH: 0/0/2/0.3.0                        MPETYPE:      24   MPESUBTYPE:  7
CLASS: TAPE     DDUMP    TAPE1

LDEV:   100  DEVNAME: 00000100           OUTDEV:        0   MODE:
   ID: PASSTHRU                           RSIZE:       128   DEVTYPE: MOSAR_AC
 PATH: 0/6/2/0.3.0                        MPETYPE:      24   MPESUBTYPE:  4
CLASS:

LDEV:   400  DEVNAME: 00000400           OUTDEV:        0   MODE:
   ID: HVDDISK                            RSIZE:       128   DEVTYPE: DISC
 PATH: 0/6/0/0.0.0                        MPETYPE:       4   MPESUBTYPE:  2
CLASS: DISC     SPOOL
```

Recent Enhancements

1. User notification of Bus Resets

Without this enhancement a bus reset (a.k.a powerfail reset, power-on reset) will result in device configuration settings (SCSI MODE SELECT) made by an application program being lost silently.

This enhancement enables the different SCSI DMs (SCSI Tape DM, SCSI Tape2 DM, SCSI Disc DM and SCSI DISC ARRAY DM) to 'remember' a bus reset and return a *device_reset* status back to the application program.

The following are some details regarding this enhancement:

- Each DM 'remembers' a bus reset when it happens. During the subsequent I/O processing, the I/O request is *not* forwarded to the lower I/O levels. The DM returns a *device_reset* status (0x800) to the application program. The next I/O request will be processed as usual.
- It is recommended that programmers always begin with a 'TUR' (Test Unit Ready) command so that any stale device reset status is flushed out. (Programming note: MPE/iX drivers typically perform the following commands: IDENTIFY, TUR, TUR, TUR, IDENTIFY to clear all possible bad status and verify a device is available).
- When there are multiple accessors (independent programs, threads, forked children/parent) of a particular DM, it should be noted that only the *first* accessor after a bus reset will return a *device_reset* status. An I/O that is in progress during the bus reset will also return a *device_reset* status.
- On receipt of a *device_reset*, the program should re-issue any MODE SELECT commands to re-set any device operating configuration.

This enhancement is available via patch MPENX03A for MPE/iX release C.75.00.

2. LDEV number to I/O path Conversion

A new user callable routine (call privilege 3, execution privilege 0) is available in MPE/iX to obtain the I/O path of a given logical device. The following are the external specifications of this procedure:
Syntax *HPLDEVTOIOPATH (ldev, iopath, status);*

Parameters

- *ldev* (required) is of type *ldev_type* which is a 16 bit unsigned value (bit16). This parameter is passed by value.

```
ldev_type = bit16;
```

- *io_path* (required) is of type pac32_t where pac32_t is a packed array of 32 characters. This parameter is passed by reference.

```
io_path = packed array[1..32] of char;
```

- *status* (required) is of type hpe_status. This *status* parameter is also passed by reference.

```
hpe_status = record
case boolean of
true : (all : integer);
false: (status : shortint;
subsys : shortint);
end;
```

Operation Notes

MPE manages devices based on an I/O tree structure similar to the output of ioscan in HP-UX. The physical path of each device establishes its hierarchy. This physical path is needed to make SPT calls to these devices. It is easier to work with devices using their unique LDEV number instead of their hierarchical physical path. It will be very handy for programmers to use this user callable routine and convert an LDEV number to its corresponding physical path.

HPLDEVTOIOPATH converts a logical device number provided in *ldev* to its physical path and returns this information in the *io_path* parameter. The path is copied left justified into *io_path* and is followed by spaces to fill the character array. *Status* indicates the outcome of the conversion. A *status* value of zero indicates that all's well. The call to HPLDEVTOIOPATH should be within a *TRY/RECOVER* block so that escapes from the routine are caught when the status parameter passed is invalid (fails the parameter checks)
Please see Appendix-B for a listing of programs invoking HPLDEVTOIOPATH. A snapshot of the results is also given. This enhancement is available via patch MPENX04A for MPE/iX release C.75.00

3. Dup capability on SPT device files

A ‘dup’ function has been introduced for SPT device files to duplicate them and thus have multiple forked processes making SPT calls. SPT calls are made as described in the previous section (please see appendix A for a program listing). Child processes can be created after opening the SPT device file using the *fork()* call. This enhancement is available via patch MPENX04A for MPE/iX release C.75.00

4. Using SPT Device files across system boots

The SPT *ioctl* call results in direct communication between the device managers (software) and SCSI devices (hardware) via the ‘port’ number of the DM instead of the unique (constant) LDEV number. A ‘Port’ is the low level medium for inter process communication in MPE/iX. There is a possibility of the port assigned to each DM getting changed with each system boot. As a consequence, these device files may not be valid across boots.

MPE/iX is now modified to create SPT device files that "store" the physical path of the corresponding device instead of the DM port number. This path is converted to the current DM port dynamically and the I/O happens. This change will make the device files valid across boots. Application Programmers, hence, need not create the SPT device file each time. The new user callable routine HPLDEVTOIOPATH can be used to map LDEV numbers to the physical path to be used for the one-time creation of the SPT device file.

CAUTION: If you change a device’s SCSI ID, or move it from one HBA to another, the physical path to that device also changes which would require creating a new device file for that new path. Further, if you then add a new device at the old location the old device file will be able to access it (unless you purge the old file at the time you moved the old device). Therefore it is a good idea to purge and rebuild device files when configuration changes of this sort are made on a system. Patch MPENX04A provides this enhancement for MPE/iX release C.75.00.

LIBIO.H

Bug fixed in /usr/include/sys/libIO.h

File /usr/include/sys/libIO.h has the following include statement at the beginning:

```
#ifndef MPEIX
#include </sys/ioparams.h>
#else
#include <ioparams.h>
#endif
```

The leading / in the path under the MPEIX inclusion is incorrect. This / makes the path absolute and file /sys/ioparams.h is non existent on MPE systems. This will hence result in a compilation error. The compilation will succeed if the leading / is removed.

The leading / has been removed and the updated header file is available via patch MPENX01A. This patch copies the header file to H.SYS and also includes a custom install job (IHFNX01) that copies this file to the HFS directory /usr/include/sys.

Appendix A

Program listing to execute a SCSI Inquiry command by SPT via TM_Diag

tmdiag.c

```
#pragma width 132
#define _POSIX_SOURCE

/* tmdiag.c                                                              */
/* Given an MPE SCSI Device Path, attempt to mknod a device file and     */
/* then open, ioctl(two times), close, unlink the file. . .              */
/*   Open with parameters that will have us go through the tm_diag TM    */
/*                                                                       */
/* Original SCSI ioctl() code written by Chris Moore of HP-UX GSC        */
/* ported from HP-UX to MPE/iX POSIX by Jim Hawkins MPE-iX Lab. . .  */
/*                                                                       */
/*                                                                       */
/*c89 -P LTMDIAG -o TMDIAG -Wc,+e,+m,+o,-C "-WL,XL=LIBIOXL.LIB.SYS" tmdiag.c */
/*                                                                       */
/* it appears on 7.5 you may have to use HFS syntax for the library      */
/*c89 -P LTMDIAG -o TMDIAG -Wc,+e,+m,+o,-C "-WL,XL=/SYS/LIB/LIBIOXL" tmdiag.c*/
/*                                                                       */
/* -P -- listing file name (MPE name space only)                         */
/* -o -- program name (MPE or posix name space)                          */
/* -Wc,   -- pass the next options to the ccxl compiler                  */
/*     +e -- turn on long pointer support                                */
/*     +m -- turn on identifier maps                                     */
/*     +o -- turn on code offsets                                        */
/*     -C -- Retain comments                                             */
/* -WL, -- pass the next option to the linker                            */
/*    XL=LIBIOXL.LIB.SYS" -- link the program with this library          */
/* tmdiag.c -- program to compile                                        */
/*                                                                       */
/* The resuling “TMDIAG” program can be run                              */
/* :RUN TMDIAG;info=”0/0/1/1.15.0”                                       */
/*                   ^^^^^ or other valid path on your system            */
/*       program. . . .                                                  */


#include <stdio.h>          /* basic terminal I/O        */
#include <errno.h>          /* for file I/O errors       */
#include <sys/libIO.h>      /* for io_search, io_query   */
#include <sys/stat.h>       /* for mknod parms           */
#include <sys/sysmacros.h>  /* for mknod "dev" macros    */
#include <fcntl.h>          /* for ioctl                 */
#include <unistd.h>         /* for ANSI/POSIX            */
#include <sys/scsi.h>       /* for SCSI stuff            */

/* global variables */
  struct sctl_io command;
  struct sense_2_aligned sense_data;
  unsigned char buf[64*1024];

/* print the command structure as hex integer values */
void print_command ( struct sctl_io x )
{ int i;
  unsigned int *j;  /* pointer to some, yet to be specified location */

  j = (unsigned int *) &x;
  for (i=0; i < sizeof (struct sctl_io); i=i+4) {
    printf ("%08x ", *j++);
    } /* for loop */
  printf ("\n");
}

void check_diag_command ( struct sctl_io command )
{
  int i;
  struct inquiry_2 *inq;  /* ptr to some inquiry data, location tbd */

  printf ("\n");
  if (command.cdb_status != S_GOOD) {
    fprintf(stderr, "SCSI command returned status 0x%02x\n",
                    command.cdb_status);
    if (command.cdb_status == S_CHECK_CONDITION) {
      if (command.sense_status != S_GOOD) { /* request sense failed? */
        fprintf(stderr, "REQUEST SENSE returned status 0x%02x\n",
                        command.sense_status);
      }
      else { /* request sense worked, print details */
        /* CHANGE TO "&sense_data" from "sense_data" */
        memcpy(&sense_data, &(command.sense), command.sense_xfer);
        fprintf(stderr, "Sense error code: 0x%02x\n", sense_data.error_code);
        fprintf(stderr, "Sense key: 0x%02x\n", sense_data.key);
        fprintf(stderr, "Sense code: 0x%02x\n", sense_data.code);
        fprintf(stderr, "Sense qualifier: 0x%02x\n", sense_data.qualifier);
      } /* else sense_status = S_GOOD */
    } /* if we have a check condition, print details */
  } /* if command status != S_GOOD */
  else {
    printf ("Good status, %d bytes returned (in hex): \n", command.data_xfer );

    for (i=0; i<command.data_xfer; i++ ) {
      printf ("%02x", buf [i]);
    } /* for loop */
    printf ("\n");

    if (command.cdb[0] == 0x12) {

      /* inq is declared as a pointer to a "struct inquiry_2" above */
      /* point it at buf[] which should contain our inquiry data  */
      inq = (struct inquiry_2 *) &buf;

      /* display the device type */
      switch  (inq->dev_type)  {
        case SCSI_DIRECT_ACCESS:
          printf ("SCSI Direct Access (Disk) \n");
          break;

        case SCSI_SEQUENTIAL_ACCESS:
          printf ("SCSI Sequential Access (Tape) \n");
          break;

        case SCSI_UNKNOWN_DEV_TYPE:
          printf ("Unknown or No Device Type \n");
          break;

        default:
          printf ("Device type not programmed: %02x \n", inq->dev_type );
          break;
      } /* end of switch */
      /* sizeof either uses "(type)" or "expression" */
      /* in this case we don't have a type for inquiry so use the expression */
      /* inq->vendor_id to get the size of the subfield of the structure     */

      printf ("Vendor: ");
      for (i=0; i < (sizeof inq->vendor_id) ;i++) {
        printf ("%c", inq->vendor_id[i]);
      } /* print vendor_id as chars */
      printf ("\n");

      printf ("Product : ");
      for (i=0; i < (sizeof inq->product_id) ;i++) {
        printf ("%c", inq->product_id[i]);
      } /* print product_id as chars */
      printf ("\n");

      printf ("Rev: ");
      for (i=0; i < (sizeof inq->rev_num) ;i++) {
        printf ("%c", inq->rev_num[i]);
      } /* print rev_num as chars */

      printf ("\n");

    } /* if inquiry command, format returned data */
  } /* else good status diplay results */
} /* check_diag_command */

main(argc, argv)
int argc;
char *argv[];
{
  char file_name[100] = "";
  char path_name[100] = "";
  char *path_ptr = path_name;
  io_token_t token;
  hw_path_t  query_hw_path;
  hw_path_t  device_hw_path;
  hw_path_t  *query_ptr = &query_hw_path;
  hw_path_t  *device_ptr = &device_hw_path;
  int        query_minor_num;
  int fd;
  int r;
  unsigned timeout;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <device path>\n", argv[0]);
    exit(1);
  }

  timeout = 10000;

  /* start of MPE changes */
  /* original code simply called open with argv[1]   */
  strcpy (path_name,  argv[1]);

  /*                                                 */
  /* In this case we're trying to get to the TM_DIAG */
  /* To do this there is a magic incantation:        */
  /*   On the mknod you pass in a slighly altered    */
  /*   version of the MPE/iX "Port Number" for the   */
  /*   device.                                       */
  /* So, start with an MPE/iX Path, validate it and  */
  /* then us io_search to get the "minor_num" for    */
  /* that path.                                      */

  (void) io_str_to_hw_path (path_ptr, device_ptr);
  if (io_errno != 0)
    {
    printf ("io_query - token: %x io_error: %x \n",
            token,
            io_errno );
    exit(5);
    }

  /* if we're here we must have a valid device path syntax */
  /* therefore io_query and io_search should work w/o major error */
  /* search the tree for a node with the same path as input */
  token = 0;   /*  start at "root" of io tree  */
  token = io_search ( token, S_IOTREE, 0, "hw_path", device_ptr);

  /* if token is null then exit the program */
  if (token == NULL)
    { printf (" io_search didn't find path matching: %s \n", path_name);
    exit (6);
    }
  else  /* get the minor number to use with mknod */
    (void) io_query (token, S_IOTREE, "minor_num", &query_minor_num);

  /* convert device path into a string good for a file name we can use */
  /* 10/4/4.12.0 ===> 10_4_4.12.0                    */
  while ( *path_ptr)
    {
    *path_ptr =  (*path_ptr == '/') ? '_' : *path_ptr ;
    path_ptr++;
    }

  strcpy (file_name, "/dev/tm_diag_");
  strcat (file_name, path_name);

  printf ("%s \n", file_name);

  /* start by unlink (delete/purge) any old file with same name */
  /* don't care too much about results. . .                     */
  r =  unlink (file_name);

  /* for tm_diag type file */
  /* mknod "character special" with dev_t using lower 24 bits of port with */
  /* zeroed  upper byte ('major'), makedev is macro from sys/sysmacros.h */
  r = mknod (file_name, S_IFCHR, makedev (0, query_minor_num));

  if (r != 0)
    {
    printf ("mknod: Error number: %d \n", errno);
    exit (7);
    }
  /* (else) no mknod problem */

  fd = open(file_name,O_RDWR);
  if (fd < 0) {
    perror("open");
    fprintf(stderr, "Can't open device file %s\n", file_name);
    exit(3);
  }

  /* do an inquiry */
  memset(command, 0, sizeof(struct sctl_io));
```

```
  command.flags = SCTL_READ;
  command.cdb_length = 6;
  command.cdb[0] = CMDinquiry;  /* Inquiry */
  command.cdb[4] = sizeof(struct inquiry_2);  /* Allocation Length */
  command.data = buf;
  command.data_length = 64*1024;
  command.max_msecs = timeout;

/*print_command ( command );*/

  r = ioctl(fd, SIOC_IO, &command);
  if (r) {
    perror("ioctl");
    fprintf(stderr, "SIOC_IO ioctl failed\n");
    exit(4);
  }

/*print_command ( command );*/
  check_diag_command (command);

  /* do an illegal command to test error stuff */
  memset(command, 0, sizeof(struct sctl_io));
  command.flags = SCTL_READ;
  command.cdb_length = 6;
  command.cdb[0] = 0x09;  /* Invalid SCSI Command */
  command.data = buf;
  command.data_length = 64*1024;
  command.max_msecs = timeout;

/*print_command ( command );*/

  r = ioctl(fd, SIOC_IO, &command);
  if (r) {
    perror("ioctl");
    fprintf(stderr, "SIOC_IO ioctl failed\n");
    exit(4);
  }

/*print_command ( command );*/
  check_diag_command (command);

  /* close the device file */
  r = close (fd);

  /* unlink (purge/delete) the device file we created */
  r = unlink (file_name);

} /* end of main */
```

Sample output

(Disk)

```
:tmdiag "0/0/1/1.15.0"
/dev/tm_diag_0_0_1_1.15.0

Good status, 128 bytes returned (in hex):
000002128b00013e53454147415445205354333138323735 4c43202020202020 4850303731414b30303
551470000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000436f707972696768742028632920323030302053656167617465204
16c6c20
SCSI Direct Access (Disk)
Vendor: SEAGATE
Product : ST318275LC
Rev: HP07

SCSI command returned status 0x02
Sense error code: 0x70
Sense key: 0x05
Sense code: 0x20
Sense qualifier: 0x00
```

(Tape)

```
:tmdiag 7 "0/0/2/0.6.0"
/dev/tm_diag_0_0_2_0.6.0

Good status, 43 bytes returned (in hex):
01800202260000184850202020202020433135333741202020202020202020204850303220203538000
002
SCSI Sequential Access (Tape)
Vendor: HP
Product : C1537A
Rev: HP02

SCSI command returned status 0x02
Sense error code: 0x70
Sense key: 0x05
Sense code: 0x20
Sense qualifier: 0x00
```

Appendix B

A simple modcal program to invoke *HPLDEVTOIOPATH*

```
{
 File: xldevtoiop

This program accepts an LDEV number and returns its I/O path. The path is returned
in an array of 32 characters. The I/O path returned is followed by spaces and hence
may need trimming before use.

Steps to link and add capabilities:

    1. PASXL xld2iop, yld2iop
    2. link yld2iop, eld2iop; cap=ba,ia,pm,ph
}
$standard_level 'ext_modcal'$
program testldevtoiopath(input,output);

type
pac32_t = packed array[1..32] of char;
ldev_type = bit16;

hpe_status = record
case boolean of
true : (all : integer);
false: (status : shortint;
subsys : shortint);
end;

var
path : pac32_t;
ldev : ldev_type;
status : hpe_status;

procedure hpldevtoiopath (
                  ldev  : ldev_type;
            var  iopath : pac32_t;
            var  status : hpe_status);external;
begin
 try
  path := '';
  status.all := 0;

  writeln('Enter LDEV number: ');
  read(ldev);
  hpldevtoiopath(ldev,path,status);

  if (status.all = 0) then
   writeln('The I/O path is: ',path)
  else
   writeln('Error status - Info: ',status.status,
             ' ,Subsys: ',status.subsys);
recover
  begin
      status.all := escapecode;
      writeln('Error status- Info: ',status.status,
  ' ,Subsys: ',status.subsys);
  end;
end.
```

Sample output

```
:eld2iop
Enter LDEV number:
20
The I/O path is: 0/0/4/1.0


:eld2iop
Enter LDEV number:
6
The I/O path is: 0/0/1/0.16.0


:eld2iop
Enter LDEV number:
7
The I/O path is: 0/0/1/0.6.0


:eld2iop
Enter LDEV number:
1
The I/O path is: 0/0/2/1.6.0


:eld2iop
Enter LDEV number:
0
Error status - Info:        -7134 ,Subsys:          143


:eld2iop
Enter LDEV number:
24
Error status - Info:        -7134 ,Subsys:          143


:eld2iop
Enter LDEV number:
321
Error status - Info:        -7134 ,Subsys:          143
```

```
$1 ($35) nmdebug > wl errmsg(-#7134,#143)
An invalid ldev number was specified or a bounds violation occurred.
```

A simple C program to invoke HPLDEVTOIOPATH

```
/*
 File: ldevtoiop.c

This program accepts an LDEV number and returns its I/O path. The path is returned
in an array of 32 characters. The I/O path returned is followed by spaces and hence
may need trimming before use.

Steps to link and add capabilities:

    1. In POSIX Shell run: c89 ldevtoiop.c -o ELD2IOP
    2. Exit shell, run linkedit
    3. altprog ELD2IOP;CAP=BA,IA,PM,PH
*/
#pragma standard_level "ext_modcal"
#pragma intrinsic GETPRIVMODE
#include <stdio.h>

typedef
struct path_str_struct
{
  char  str[32];
} path_str_t;

typedef union
{
 int   all;
 struct {
          short info;
          short subsys;
         }   part;
 } hpe_status;

typedef short ldev_type;

extern void hpldevtoiopath( ldev_type ldev, path_str_t* path, hpe_status* sts);

void main()
{
  path_str_t path;
  ldev_type  ldev;
  hpe_status status;
  int        ldev_temp;

  ldev = 7;
  status.all = 0;

  printf("Please enter the ldev no. for which you need the I/O path: ");
  scanf("%d",&ldev_temp);
  ldev = ldev_temp;

  GETPRIVMODE();
  hpldevtoiopath (ldev, &path, &status);

  if (status.all != 0)
    printf ("Error Status returned: info %d subsys %d\n",
              status.part.info, status.part.subsys);
  else
   printf ("IO Path returned is %s\n",path);
}
```

Sample output

```
:eld2iop
Please enter the ldev no. for which you need the I/O path: 20
IO Path returned is 0/0/4/1.0

:eld2iop
Please enter the ldev no. for which you need the I/O path: 6
IO Path returned is 0/0/1/0.16.0

:eld2iop
Please enter the ldev no. for which you need the I/O path: 7
IO Path returned is 0/0/1/0.6.0

:eld2iop
Please enter the ldev no. for which you need the I/O path: 1
IO Path returned is 0/0/2/1.6.0

:eld2iop
Please enter the ldev no. for which you need the I/O path: 0
Error Status returned: info -7134 subsys 143

:eld2iop
Please enter the ldev no. for which you need the I/O path: 24
Error Status returned: info -7134 subsys 143

:eld2iop
Please enter the ldev no. for which you need the I/O path: 321
Error Status returned: info -7134 subsys 143
```

Appendix C

CR numbers and Patch IDs *(all patches are for MPE/iX release C.75.00)*

DESCRIPTION	CR NUMBER	PATCH ID
User notification of Bus Reset	JAGag36874	MPENX03A
User callable routine to convert LDEV to IOPATH	JAGag36877	MPENX04A
TM_DIAG dup capability	JAGag36987	MPENX04A
Using TM_DIAG device files across boots	JAGag36989	MPENX04A
Fix for libIO.h bug	JAGag36873	MPENX01A