General Standards Corporation
8302A Whitesburg Drive
Huntsville, AL 36802
Phone (256) 880-8787
Fax (256) 880-8788

General Standards Corporation

PCI-HPDI-32 Windows NT Device Driver User's Manual

Windows NT Device Driver Software for the General Standards PCI-HPDI-32 hosted on x86 Processors

Document number:	?	Revision:	1.0	Date: 2/03/99
Engineering Approval:				Date:
Quality Representative Approval:				Date:

Acknowledgments

1. Scope

The purpose of this document is to describe how to interface with the PCI-HPDI-32 Windows NT Device Driver developed by General Standards Corporation (GSC). This software provides the interface between "Application Software" and the PCI-HPDI-32 Board. The interface to this board is at the device level. It requires no knowledge of the actual board addressing or device register contents.

The PCI-HPDI-32 Driver Software executes under control of the Windows NT operating system. The PCI-HPDI-32 is implemented as a standard Windows NT device driver written in the 'C' programming language. The PCI-HPDI-32 Driver Software is designed to operate on CPU boards containing standard x86 processors.

2. Hardware Overview

The General Standards Corporation (GSC) PCI-HPDI-32 board is a high-speed parallel digital interface that fits into a PCI Card slot. This board provides for bi-directional data transfers between two computers, or one computer and an external peripheral, of up to 100 Mbytes per second. This board also can transfer data indefinitely without host intervention. Once the data link between the two computers is established, the desired transfers can be performed and will become transparent to the user.

The PCI-HPDI-32 board includes a DMA controller, 64/128/256/512 Kbytes of FIFO buffering, a cable input/output controller, and cable transceivers (RS-422/485 or differential pseudo ECL). The DMA controller is capable of transferring data to and from host memory using 32 bit transfers; whereas the FIFO memory provides a means for continuous transmission of data without interrupting the DMA transfers or requiring intervention from the host CPU. The board also provides for DMA chaining, interrupt generation for various states of the board, including but not limited to end-of-transfer, FIFO empty, FIFO almost empty, FIFO almost full, and FIFO full.

The configuration of the interrupting capability of the PCI-HPDI-32 board is described in the hardware manual for the board. The PCI-HPDI-32 Device Driver must be used correctly in accordance with the hardware configuration in order to provide consistent results.

3. Referenced Documents

The following documents provide reference material for the PCI-HPDI-32 Board:

- PCI-HPDI-32 User Manual - Preliminary Version N/R, General Standards Corporation.
- PLX Technology, Inc. PCI 9080 PCI Bus Master Interface Chip data sheet.

4. Driver Interface

The PCI-HPDI-32 Driver conforms to the device driver standards required by the Windows NT Operating System and contains the following standard driver entry points.

- CreateFile() - opens a driver interface to one PCI-HPDI-32 Card
- CloseHandle() - closes a driver interface to one PCI-HPDI-32 Card
- ReadFile() - reads data received from a PCI-HPDI-32 Card
- WriteFile() - writes data to be transmitted by a PCI-HPDI-32 Card
- DeviceIoControl() - performs various control and setup functions on the PCI-HPDI-32 Card

The PCI-HPDI-32 Device Driver provides a standard driver interface to the GSC PCI-HPDI-32 card for Windows NT applications which run on a x86 target processor. The device driver is installed and devices are created when the driver is started when the computer boots. The functions of the driver can then be used to access the board. Devices are created with the name "hpdix" where 'x' is the device number. Device numbers start at 0 for Windows NT applications and 1 for DOS applications. For each PCI-HPDI-32 board found, the device number increments.

Included in the device driver software package is a menu driven board application program and source code. This program is delivered undocumented and unsupported but may be used to exercise the PCI-HPDI-32 card and device driver. It can also be used to break the learning curve somewhat for programming the PCI-HPDI-32 device.

It is important to note that the PCI-HPDI-32 device driver is target processor dependent. System calls are made within the driver which are only available on x86 processors.

When the driver is installed during the power up of the computer, certain default values are set by the driver. These values are not reset to the default values every time the user calls the CreateFile function. The following default values are set:

- Read Timeout = 2 seconds
- Write Timeout = 2 seconds
- TX Almost Full = 0x0400
- TX Almost Empty = 0x0010
- RX Almost Full = 0x0400
- RX Almost Empty = 0x0010
- TX Status Length = 0
- TX Row Valid Length = 0
- TX Row Invalid Length = 0

4.1. CreateFile()

The CreateFile() function is the standard NT entry point to open a connection to a PCI-HPDI-32 Card. This function must be called before any other driver function may be called to control the device. The fdwAttrsAndFlags parameter needs to be set to FILE_FLAG_OVERLAPPED if overlapped I/O is to be performed. Using overlapped I/O for reads and writes allows the calling task to continue executing while the driver is performing the I/O operation and making calls to GetOverlappedResult() to determine when the operation is complete. See the HPDITest sample code for a example on how to perform overlapped I/O.

Multiple tasks may call CreateFile to access the driver for the same board. The programmer needs to be very careful if this is desirable. One task may set values that conflict with the other task.

PROTOTYPE:

```
HANDLE CreateFile(LPCTSTR                lpszName,
                  DWORD                  fdwAccess,
                  DWORD                  fdwShareMode,
                  LPSECURITY_ATTRIBUTES  lpsa,
                  DWORD                  fdwCreate,
                  DWORD                  fdwAttrsAndFlags,
                  HANDLE                 hTemplateFile)
```

Where:

lpszName - name of the device being opened which is "hpdix" where x is the device number. Device numbers begin with 0 for NT applications and 1 for DOS applications. Each device is consecutively numbered after that.

fdwAccess – a logically or'ed combination of one or more of the following access modes:

GENERIC_ALL	- Execute, Read and Write Access
GENERIC_EXECUTE	- Execute Access
GENERIC_READ	- Read Access
GENERIC_WRITE	- Write Access

Use GENERIC_WRITE, GENERIC_READ or (GENERIC_WRITE | GENERIC_READ) for this parameter.

fdwShareMode – a logically or'ed combination of zero or more of the following share options:

FILE_SHARE_READ	- Read Share Mode

FILE_SHARE_WRITE - Write Share Mode

This parameter is usually zero.

lpsa – a pointer to a security attributes structure.

This parameter is usually NULL.

fdwCreate – a logically or'ed combination of one or more of the following device creation options:

CREATE_NEW	- Create new file
CREATE_ALWAYS	- Always create file
OPEN_EXISTING	- Open existing file/device
OPEN_ALWAYS	- Always open file
TRUNCATE_EXISTING	- Truncate file

Use OPEN_EXISTING for this parameter.

fdwAttrsAndFlags – a logically or'ed combination of zero or more of the following attributes and flags:

FILE_ATTRIBUTE_READONLY	- Read-only file/device
FILE_ATTRIBUTE_HIDDEN	- Hidden file
FILE_ATTRIBUTE_SYSTEM	- System file
FILE_ATTRIBUTE_DIRECTORY	- Directory file
FILE_ATTRIBUTE_ARCHIVE	- Archive file
FILE_ATTRIBUTE_NORMAL	- Normal file/device
FILE_ATTRIBUTE_TEMPORARY	- Temporary file
FILE_FLAG_WRITE_THROUGH	- Write through access
FILE_FLAG_OVERLAPPED	- Overlapped access
FILE_FLAG_NO_BUFFERING	- No buffering
FILE_FLAG_RANDOM_ACCESS	- Random Access
FILE_FLAG_SEQUENTIAL_SCAN	- Sequential Scan
FILE_FLAG_DELETE_ON_CLOSE	- Delete file on Close
FILE_FLAG_BACKUP_SEMANTICS	- Backup semantics
FILE_FLAG_POSIX_SEMANTICS	- POSIX semantics
FILE_FLAG_OPEN_REPARSE_POINT	- Open reparse point
FILE_FLAG_OPEN_NO_RECALL	- Open no recall

Use either FILE_ATTRIBUTE_NORMAL or FILE_ATTRIBUTE_OVERLAPPED for this parameter. Use FILE_ATTRIBUTE_OVERLAPPED if you plan to use overlapped I/O.

hTemplateFile – handle to a template file.

Use NULL for this parameter.

Returns a handle to the device opened on success. This handle is then used as a parameter to all other device accesses. Returns a NULL when the create fails.

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;


/*   Open the PCI-HPDI-32 device hpdi1   */
hDevice = CreateFile("\\\\.\\hpdi1", GENERIC_READ | GENERIC_WRITE, 0,
                     NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
if (hDevice == INVALID_HANDLE_VALUE)
{
   dwErrorCode = GetLastError();
   ErrorMessage("CreateFile", dwErrorCode);
   ExitProcess(dwErrorCode);
}

/* Access the device here. */

/* Close the device here. */
```

4.2. CloseHandle()

The CloseHandle() function is the driver entry point to close a connection to a PCI-HPDI-32 Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-HPDI-32 Card. The CloseHandle() function closes an interface to a PCI-HPDI-32 device.

If multiple tasks have created connections to the driver using the CreateFile function, the CloseHandle call will only close the connection for that particular task.

PROTOTYPE:

```
BOOL CloseHandle(HANDLE hObject);
```

Where:

hObject - handle to the device to close. The return parameter from a CreateFile() call.

Returns TRUE if successful or FALSE if unsuccessful.

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;

/* Open the device and get hDevice here. */

/* Access the device here. */

/*  Close the HPDI Device.  */
if  (! CloseHandle(hDevice))
{
   dwErrorCode = GetLastError();
   ErrorMessage("CloseHandle", dwErrorCode);
}
hDevice = NULL;
```

4.3. ReadFile()

The ReadFile() function is the driver entry point to receive data from a PCI-HPDI-32 Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-HPDI-32 Card. The ReadFile() function reads the number of bytes from the receive FIFO that have been currently received up to the number given in the **NumberOfBytesToRead** parameter. Note that this parameter is the number of **bytes**, not the number of **words**. Therefore, the parameter should be a multiple of four. If the device was selected for overlapped I/O in the CreateFile() call, this function may return without any words being read and a call to the GetOverlappedResult() needs to be made to determine when the operation completes.

If multiple tasks try to access the ReadFile function at the same time, the driver will process each request in the order they are received. This function will perform programmed I/O or DMA transfers depending upon whether DMA is enabled using the IOCTL_HPDI_SET_DMA_ENABLE ioctl() function call. If DMA is enabled the DMA transfers will use demand mode DMA if it is enabled using the IOCTL_HPDI_SET_DEMAND_MODE_ENABLE ioctl() function call. Demand mode DMA is used to enable throttling DMA read operations when the FIFO is almost empty.

PROTOTYPE:

```
BOOL ReadFile( HANDLE          hFile,
               LPVOID          lpBuffer,
               DWORD           NumberOfBytesToRead,
               LPDWORD         lpNumberOfBytesRead,
               LPOVERLAPPED    lpOverlapped);
```

Where:

hFile - handle to the device returned from CreateFile()

lpBuffer - pointer to a buffer to store the data read

NumberOfBytesToRead - number of bytes to read

lpNumberOfBytesRead - pointer to a location to return the number of bytes read

lpOverlapped - pointer to an overlapped structure. If device was opened using overlapped I/O, this structure is required. If device was not opened using overlapped I/O, this parameter should be NULL. See WIN32 documentation for a structure definition and the HPDITest Program for an example of how to use it.

Returns TRUE on success. Returns FALSE on failure or if I/O is still pending on an overlapped I/O operation.

EXAMPLE:

```
#define MAXWORDS 80
HANDLE   hDevice;
ULONG    ulBuffer[MAXWORDS];
DWORD    dwBytesRead = 0;
DWORD    dwErrorCode;
BOOL     status;

/*  Read from the PCI-HPDI-32 device  */
status = ReadFile(hDevice, ulBuffer, MAXWORDS*4, &dwBytesRead, NULL);
if (! status)
{
   dwErrorCode = GetLastError();
   ErrorMessage("ReadFile", dwErrorCode);
}
else
{
   if (dwBytesRead != (MAXWORDS*4))
   {
      printf("Only read %d bytes\n", dwBytesRead);
   }
   else
   {
      /* Data read OK.  Use the data here. */
   }
}
```

4.4. WriteFile()

The WriteFile() function is the driver entry point to transmit data to a PCI-HPDI-32 Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-HPDI-32 Card. The WriteFile() function writes the number of bytes requested by the **nNumberOfBytesToWrite** parameter to the transmit FIFO. Note that this parameter is the number of **bytes**, not the number of **words**. Therefore, the parameter should be a multiple of four. If the device was selected for overlapped I/O in the CreateFile() call, this function may return without any words being written and a call to the GetOverlappedResult() needs to be made to determine when the operation completes.

If multiple tasks try to access the WriteFile function at the same time, the driver will process each request in the order they are received. This function will perform programmed I/O or DMA transfers depending upon whether DMA is enabled using the IOCTL_HPDI_SET_DMA_ENABLE ioctl() function call. If DMA is enabled the DMA transfers will use demand mode DMA if it is enabled using the IOCTL_HPDI_SET_DEMAND_MODE_ENABLE ioctl() function call. Demand mode DMA is used to enable throttling DMA write operations when the FIFO is almost full.

PROTOTYPE:

```
BOOL WriteFile(HANDLE          hFile,
               PCVOID          lpBuffer,
               DWORD           nNumberOfBytesToWrite,
               PDWORD          lpNumberOfBytesWritten,
               POVERLAPPED     lpOverlapped);
```

Where:

hFile - handle to the device returned from CreateFile()

lpBuffer - pointer to a buffer containing the data to be written

nNumberOfBytesToWrite - number of bytes to write

lpNumberOfBytesWritten - pointer to a location to return the number of bytes written

lpOverlapped - pointer to an overlapped structure. If device was opened using overlapped I/O, this structure is required. If device was not opened using overlapped I/O, this parameter should be NULL. See WIN32 documentation for a structure definition and the HPDITest Program for an example of how to use it.

Returns TRUE on success. Returns FALSE on failure or if I/O is still pending on an overlapped I/O operation.

EXAMPLE:

```
#define MAXWORDS 80
HANDLE   hDevice;
ULONG    ulBuffer[MAXWORDS];
DWORD    dwBytesWritten = 0;
DWORD    dwErrorCode;
BOOL     status;

/*  Write to the PCI-HPDI-32 device  */
status = WriteFile(hDevice, ulBuffer, MAXWORDS*4,
                   &dwBytesWritten, NULL);
if (! Status)
{
   dwErrorCode = GetLastError();
   ErrorMessage("WriteFile", dwErrorCode);
}
else
{
   if (dwBytesWritten != (MAXWORDS*4))
   {
      printf("Only Wrote %d bytes\n", dwBytesWritten);
   }
   else
   {
      /* Data written OK. */
   }
}
```

4.5. DeviceIoControl()

The DeviceIoControl() function is the device entry point to perform control and setup operations on a PCI-HPDI-32 Card. This function should only be called after the CreateFile() function has been successfully called for a PCI-HPDI-32 Card. The DeviceIoControl() function will perform different functions based upon the dwIoControlCode parameter. These functions will be described in the following subparagraphs.

Certain DeviceIoControl function calls should not be used unless absolutely necessary. These routines are provided so that the driver is complete and does not limit the use of the PCI-HPDI-32 board. Each of the subsections that follow will describe the limitations on their use.

PROTOTYPE:

```
BOOL DeviceIoControl(HANDLE          hDevice,
                     DWORD           dwIoControlCode,
                     LPVOID          lpInBuffer,
                     DWORD           nInBufferSize,
                     LPVOID          lpOutBuffer,
                     DWORD           nOutBufferSize,
                     LPDWORD         lpBytesReturned,
                     LPOVERLAPPED    lpOverlapped);
```

Where:

hDevice - handle to the device returned from CreateFile()

dwIoControlCode - control code for the operation to perform. One of the following constants from the include file "**HPDIIoctl.h**":

- IOCTL_HPDI_NO_COMMAND
- IOCTL_HPDI_READ_REGISTER
- IOCTL_HPDI_WRITE_REGISTER
- IOCTL_HPDI_RESET_BOARD
- IOCTL_HPDI_RESET_FIFO
- IOCTL_HPDI_ENABLE_TRANSFER
- IOCTL_HPDI_START_STOP_TRANSFER
- IOCTL_HPDI_CONFIG_TRANSFER
- IOCTL_HPDI_READ_BOARD_STATUS
- IOCTL_HPDI_READ_RX_STATUS_LEN_CTR
- IOCTL_HPDI_READ_RX_ROW_VAL_LEN_CTR
- IOCTL_HPDI_GET_DEVICE_ERROR
- IOCTL_HPDI_READ_PCI_CONFIG
- IOCTL_HPDI_READ_LOCAL_CONFIG
- IOCTL_HPDI_WRITE_PCI_CONFIG_REG

- IOCTL_HPDI_WRITE_LOCAL_CONFIG_REG
- IOCTL_HPDI_SET_READ_TIMEOUT
- IOCTL_HPDI_SET_WRITE_TIMEOUT
- IOCTL_HPDI_SET_DMA_ENABLE
- IOCTL_HPDI_SET_DEMAND_MODE_ENABLE

lpInBuffer	- pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not require input data. See the individual subsections for a description of the structures required.
nInBufferSize	- size, in bytes, of the buffer pointed to by lpInBuffer
lpOutBuffer	- pointer to a buffer that receives the operation's output data. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not produce output data. See the individual subsections for a description of the structures required.
nOutBufferSize	- size, in bytes, of the buffer pointed to by lpOutBuffer
lpBytesReturned	- pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutbuffer.
lpOverlapped	- pointer to an overlapped structure. No DeviceIoControl calls use overlapped I/O. This parameter is passed as NULL.

Returns TRUE if successful or FALSE if unsuccessful.

4.5.1. IOCTL_HPDI_NO_COMMAND

This is an empty driver entry point. This command may be given to validate that the driver is correctly installed and that the PCI-HPDI-32 Board Device has been successfully opened.

Input/Output Buffer:

not used

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwErrorCode;
DWORD    dwTransferSize;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_NO_COMMAND, NULL, 0,
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.2. IOCTL_HPDI_READ_REGISTER

The IOCTL_HPDI_READ_REGISTER Function reads and returns the contents of one of the PCI-HPDI-32 Registers.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_REGISTER_PARAMS
{
    ULONG eHPDIRegister;
    ULONG ulRegisterValue;
} HPDI_REGISTER_PARAMS, *PHPDI_REGISTER_PARAMS;

Where ulRegisterValue will store the value read from the register
and eHPDIRegister is one of the following:

#define FIRM_REV_REG           0
#define BOARD_CTRL_REG         1
#define BOARD_STATUS_REG       2
#define TX_PROG_ALMOST_REG     3
#define RX_PROG_ALMOST_REG     4
#define TX_RX_FIFO_REG         6
#define TX_STATUS_LEN_REG      7
#define TX_ROW_VAL_LEN_REG     8
#define TX_ROW_INV_LEN_REG     9
#define RX_STATUS_LEN_CTR_REG 10
#define RX_ROW_LEN_CTR_REG    11
#define INT_CTRL_REG          12
#define INT_STATUS_REG        13
```

EXAMPLE:

```
HANDLE                hDevice;
DWORD                 dwTransferSize;
DWORD                 dwErrorCode;
HPDI_REGISTER_PARAMS  InputRegData;
HPDI_REGISTER_PARAMS  OutputRegData;

InputRegData.eHPDIRegister    = FIRM_REV_REG;
OutputRegData.ulRegisterValue = 0xDEADBEEF;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_REGISTER,
                       &InputRegData, sizeof(HPDI_REGISTER_PARAMS),
                       &OutputRegData, sizeof(HPDI_REGISTER_PARAMS),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_REGISTER_PARAMS)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Firware/Revision Register = %08lx\n",
          OutputRegData.ulRegisterValue);
}
```

4.5.3. IOCTL_HPDI_WRITE_REGISTER

The IOCTL_HPDI_WRITE_REGISTER Function writes a value to one of the PCI-HPDI-32 Registers. The user should be very careful modifying values of certain registers. This is because they are used by the driver's interrupt handler to process receive and/or transmit data. The following registers should not be changed:

- INT_CTRL_REG
- INT_STATUS_REG

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_REGISTER_PARAMS
{
    ULONG eHPDIRegister;
    ULONG ulRegisterValue;
} HPDI_REGISTER_PARAMS, *PHPDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eHPDIRegister is one of the following:

#define BOARD_CTRL_REG         1
#define TX_PROG_ALMOST_REG     3
#define RX_PROG_ALMOST_REG     4
#define TX_RX_FIFO_REG         6
#define TX_STATUS_LEN_REG      7
#define TX_ROW_VAL_LEN_REG     8
#define TX_ROW_INV_LEN_REG     9
#define INT_CTRL_REG          12
#define INT_STATUS_REG        13
```

EXAMPLE:

```
HANDLE                hDevice;
DWORD                 dwTransferSize;
DWORD                 dwErrorCode;
HPDI_REGISTER_PARAMS  InputRegData;

InputRegData.eHPDIRegister   = TX_PROG_ALMOST_REG;
InputRegData.ulRegisterValue = 0x10101010;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_WRITE_REGISTER,
                      &InputRegData,  sizeof(HPDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.4. IOCTL_HPDI_RESET_BOARD

The IOCTL_HPDI_RESET_BOARD function will command the PCI-HPDI-32 Board to reset. The user should not access this function while data transfers are in progress.

Input/Output Buffer:

not used

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_RESET_BOARD, NULL, 0,
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.5. IOCTL_HPDI_RESET_FIFO

The IOCTL_HPDI_RESET_FIFO function will reset one or both of the PCI-HPDI-32 FIFOs. The user should not access this function while data transfers are in progress.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_TX_RX_PARAMS
{
   ULONG eWhichFIFO;
} HPDI_TX_RX_PARAMS, *PHPDI_TX_RX_PARAMS;

where eWhichFIFO is one of the following:

#define TX                0
#define RX                1
#define TX_AND_RX         2
```

EXAMPLE:

```
HANDLE            hDevice;
DWORD             dwTransferSize;
DWORD             dwErrorCode;
HPDI_TX_RX_PARAMS FIFOData;

FIFOData.eWhichFIFO = TX_AND_RX;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_RESET_FIFO,
                      &FIFOData, sizeof(HPDI_TX_RX_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.6. IOCTL_HPDI_ENABLE_TRANSFER

The IOCTL_HPDI_ENABLE_TRANSFER function will enable or disable PCI-HPDI-32 transfers for receive, transmit or both.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_TX_RX_PARAMS
{
   ULONG eWhichFIFO;
} HPDI_TX_RX_PARAMS, *PHPDI_TX_RX_PARAMS;

where eWhichFIFO is one of the following:

#define TX                0
#define RX                1
#define TX_AND_RX         2
#define NEITHER_TX_NOR_RX 3
```

EXAMPLE:

```
HANDLE            hDevice;
DWORD             dwTransferSize;
DWORD             dwErrorCode;
HPDI_TX_RX_PARAMS EnableData;

EnableData.eWhichFIFO = TX_AND_RX;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_ENABLE_TRANSFER,
                      &EnableData, sizeof(HPDI_TX_RX_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.7. IOCTL_HPDI_START_STOP_TRANSFER

The IOCTL_HPDI_START_STOP_TRANSFER function either enables or disables PCI-HPDI-32 Transfers. This function is only necessary if the application is manually writing to the FIFO register. The driver automatically starts the transfer when using the WriteFile driver call.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_START_STOP_TRANS_PARAMS
{
   BOOLEAN bStartTransfer;
} HPDI_START_STOP_TRANS_PARAMS, *PHPDI_START_STOP_TRANS_PARAMS;

Where bStartTransfer is TRUE to start a transfer and FALSE to stop a
transfer.
```

EXAMPLE:

```
HANDLE                          hDevice;
DWORD                           dwTransferSize;
DWORD                           dwErrorCode;
HPDI_START_STOP_TRANS_PARAMS    StartStopData;

StartStopData.bStartTransfer = TRUE;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_START_STOP_TRANSFER,
                      &StartStopData,
                      sizeof(HPDI_START_STOP_TRANS_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.8. IOCTL_HPDI_CONFIG_TRANSFER

The IOCTL_HPDI_CONFIG_TRANSFER function initializes the PCI-HPDI-32 Board for Data Transfers. The following registers/bits will be written:

- Transmit Almost Empty Register
- Transmit Almost Full Register
- Transmit Status Length Register
- Transmit Row Length Register
- Transmit Invalid Length Register
- Receive Almost Empty Register
- Receive Almost Full Register
- Enable External Cable Start Bit

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_CONFIG_TRANSFER
{
    USHORT    uhTXAlmostEmpty;
    USHORT    uhTXAlmostFull;
    ULONG     ulTXStatusLength;
    ULONG     ulTXRowLength;
    USHORT    uhTXInvalidLength;
    USHORT    uhRXAlmostEmpty;
    USHORT    uhRXAlmostFull;
    BOOLEAN   bEnableExtCableStart;
} HPDI_CONFIG_TRANSFER, *PHPDI_CONFIG_TRANSFER;
```

EXAMPLE:

```
HANDLE                hDevice;
DWORD                 dwTransferSize;
DWORD                 dwErrorCode;
HPDI_CONFIG_TRANSFER  ConfigData;

ConfigData.uhTXAlmostEmpty      = 0x0100;
ConfigData.uhTXAlmostFull       = 0x0100;
ConfigData.ulTXStatusLength     = 1;
ConfigData.ulTXRowLength        = 1;
ConfigData.uhTXInvalidLength    = 1;
ConfigData.uhRXAlmostEmpty      = 0x0100;
ConfigData.uhRXAlmostFull       = 0x0100;
ConfigData.bEnableExtCableStart = FALSE;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_CONFIG_TRANSFER,
                      &ConfigData,  sizeof(HPDI_CONFIG_TRANSFER),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.9. IOCTL_HPDI_READ_BOARD_STATUS

The IOCTL_HPDI_READ_BOARD_STATUS function will read and return the value of the PCI-HPDI-32 Board Status Register.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_READ_STATUS_PARAM
{
   ULONG ulBoardStatus;
} HPDI_READ_STATUS_PARAM, *PHPDI_READ_STATUS_PARAM;
```

EXAMPLE:

```
HANDLE                  hDevice;
DWORD                   dwTransferSize;
DWORD                   dwErrorCode;
HPDI_READ_STATUS_PARAM  BoardStatus;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_BOARD_STATUS, NULL,
                       0, &BoardStatus,
                       sizeof(HPDI_READ_STATUS_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_READ_STATUS_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Board Status = %08lx\n", BoardStatus.ulBoardStatus);
}
```

4.5.10. IOCTL_HPDI_READ_RX_STATUS_LEN_CTR

The IOCTL_HPDI_READ_RX_STATUS_LEN_CTR function reads and returns the value of the PCI-HPDI-32 Receive Status Length Counter Register.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_READ_RX_STATUS_LEN_CTR_PARAM
{
   ULONG ulRxStatusLenCtr;
} HPDI_READ_RX_STATUS_LEN_CTR_PARAM,
  *PHPDI_READ_RX_STATUS_LEN_CTR_PARAM;
```

EXAMPLE:

```
HANDLE                                hDevice;
DWORD                                 dwTransferSize;
DWORD                                 dwErrorCode;
HPDI_READ_RX_STATUS_LEN_CTR_PARAM     RxStatusLengthCounter;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_RX_STATUS_LEN_CTR,
                       NULL, 0, &RxStatusLengthCounter,
                       sizeof(HPDI_READ_RX_STATUS_LEN_CTR_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_READ_RX_STATUS_LEN_CTR_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("RX Status Length Counter = %08lx\n",
          RxStatusLengthCounter.ulRxStatusLenCtr);
}
```

4.5.11. IOCTL_HPDI_READ_RX_ROW_VAL_LEN_CTR

The IOCTL_HPDI_READ_RX_ROW_VAL_LEN_CTR function reads and returns the value of the PCI-HPDI-32 Receive Row Valid Length Counter Register.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM
{
   ULONG ulRxRowValLenCtr;
} HPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM,
  *PHPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM;
```

EXAMPLE:

```
HANDLE                                hDevice;
DWORD                                 dwTransferSize;
DWORD                                 dwErrorCode;
HPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM  RxRowValidLengthCounter;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_RX_ROW_VAL_LEN_CTR,
                       NULL, 0, &RxRowValidLengthCounter,
                       sizeof(HPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_READ_RX_ROW_VAL_LEN_CTR_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("RX Row Valid Length Counter = %08lx\n",
          RxRowValidLengthCounter.ulRxRowValLenCtr);
}
```

4.5.12. IOCTL_HPDI_GET_DEVICE_ERROR

The IOCTL_HPDI_GET_DEVICE_ERROR function will return the error that occurred on the last call to one of the PCI-HPDI-32 Device Driver entry points. Whenever a driver function is called and it returns an error, this function may be called to determine the cause of the error.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_GET_DEVICE_ERROR_PARAM
{
   ULONG ulHPDIErrorCode;
} HPDI_GET_DEVICE_ERROR_PARAM, *PHPDI_GET_DEVICE_ERROR_PARAM;
```

EXAMPLE:

```
HANDLE                                hDevice;
DWORD                                 dwTransferSize;
DWORD                                 dwErrorCode;
IOCTL_HPDI_GET_DEVICE_ERROR_PARAM     DeviceError;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_GET_DEVICE_ERROR, NULL,
                       0, &DeviceError,
                       sizeof(HPDI_GET_DEVICE_ERROR_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_GET_DEVICE_ERROR_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Device Error = %s\n",
          pszDeviceError[DeviceError.ulHPDIErrorCode]);
}
```

4.5.13. IOCTL_HPDI_READ_PCI_CONFIG

The IOCTL_HPDI_READ_PCI_CONFIG function will read all of the PCI Configuration Registers.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_READ_PCI_CONFIG_PARAM
{
   ULONG ulDeviceVendorID;
   ULONG ulStatusCommand;
   ULONG ulClassCodeRevisionID;
   ULONG ulBISTHdrTypeLatTimerCacheLineSize;
   ULONG ulRuntimeRegAddr;
   ULONG ulConfigRegAddr;
   ULONG ulPCIBaseAddr2;
   ULONG ulPCIBaseAddr3;
   ULONG ulUnusedBaseAddr1;
   ULONG ulUnusedBaseAddr2;
   ULONG ulCardbusCISPtr;
   ULONG ulSubsystemVendorID;
   ULONG ulPCIRomAddr;
   ULONG ulReserved1;
   ULONG ulReserved2;
   ULONG ulMaxLatMinGntIntPinIntLine;
} HPDI_READ_PCI_CONFIG_PARAM, *PHPDI_READ_PCI_CONFIG_PARAM;
```

EXAMPLE:

```
HANDLE                      hDevice;
DWORD                       dwTransferSize;
DWORD                       dwErrorCode;
HPDI_READ_PCI_CONFIG_PARAM  ConfigRegs;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_PCI_CONFIG, NULL, 0,
                       &ConfigRegs,
                       sizeof(HPDI_READ_PCI_CONFIG_PARAM),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(HPDI_READ_PCI_CONFIG_PARAM)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("Device ID/Vendor ID Reg = %08lx\n",
          ConfigRegs.ulDeviceVendorID);
   printf("Status/Command Reg = %08lx\n",
          ConfigRegs.ulStatusCommand);
   printf("Class Code/Revision ID Reg = %08lx\n",
          ConfigRegs.ulClassCodeRevisionID);
   printf("BIST/Header Type/Lat Timer/Cache Line Size Reg = 08lx\n",
          ConfigRegs.ulBISTHdrTypeLatTimerCacheLineSize);
   printf("Runtime Register Address Reg = %08lx\n",
          ConfigRegs.ulRuntimeRegAddr);
   printf("Config Register Address Reg = %08lx\n",
          ConfigRegs.ulConfigRegAddr);
   printf("PCI Base Address 2 Reg = %08lx\n",
          ConfigRegs.ulPCIBaseAddr2);
   printf("PCI Base Address 3 Reg = %08lx\n",
          ConfigRegs.ulPCIBaseAddr3);
   printf("Unused Base Address 1 Reg = %08lx\n",
          ConfigRegs.ulUnusedBaseAddr1);
   printf("Unused Base Address 2 Reg = %08lx\n",
          ConfigRegs.ulUnusedBaseAddr2);
   printf("Cardbus CIS Pointer Reg = %08lx\n",
          ConfigRegs.ulCardbusCISPtr);
   printf("Subsystem ID/Vendor ID Reg = %08lx\n",
          ConfigRegs.ulSubsystemVendorID);
   printf("PCI Rom Address Reg = %08lx\n",
          ConfigRegs.ulPCIRomAddr);
   printf("Reserved 1 Reg = %08lx\n",
          ConfigRegs.ulReserved1);
   printf("Reserved 2 Reg = %08lx\n",
          ConfigRegs.ulReserved2);
   printf("Max Lat/Min Gnt/Int Pin/Int Line Reg = %08lx\n",
          ConfigRegs.ulMaxLatMinGntIntPinIntLine);
}
```

4.5.14. IOCTL_HPDI_READ_LOCAL_CONFIG

The IOCTL_HPDI_READ_LOCAL_CONFIG function will read and return the local configuration registers.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _CONFIG_REGS_PARAMS
{
    /*** Local Configuration Registers ***/
    ULONG   ulPciLocRange0;
    ULONG   ulPciLocRemap0;
    ULONG   ulModeArb;
    ULONG   ulEndianDescr;
    ULONG   ulPciLERomRange;
    ULONG   ulPciLERomRemap;
    ULONG   ulPciLBRegDescr0;
    ULONG   ulLocPciRange;
    ULONG   ulLocPciMemBase;
    ULONG   ulLocPciIOBase;
    ULONG   ulLocPciRemap;
    ULONG   ulLocPciConfig;
    ULONG   ulOutPostQIntStatus;
    ULONG   ulOutPostQIntMask;
    UCHAR   uchReserved1[8];

    /*** Shared Run Time Registers ***/
    ULONG   ulMailbox[8];
    ULONG   ulPciLocDoorBell;
    ULONG   ulLocPciDoorBell;
    ULONG   ulIntCntrlStat;
    ULONG   ulRunTimeCntrl;
    ULONG   ulDeviceVendorID;
    ULONG   ulRevisionID;
    ULONG   ulMailboxReg0;
    ULONG   ulMailboxReg1;

    /*** Local DMA Registers ***/
    ULONG   ulDMAMode0;
    ULONG   ulDMAPCIAddress0;
    ULONG   ulDMALocalAddress0;
    ULONG   ulDMAByteCount0;
    ULONG   ulDMADescriptorPtr0;
    ULONG   ulDMAMode1;
    ULONG   ulDMAPCIAddress1;
    ULONG   ulDMALocalAddress1;
    ULONG   ulDMAByteCount1;
    ULONG   ulDMADescriptorPtr1;
    ULONG   ulDMACmdStatus;
    ULONG   ulDMAArbitration;
    ULONG   ulDMAThreshold;
```

```
    UCHAR   uchReserved3[12];

    /*** Messaging Queue Registers ***/
    ULONG   ulMsgUnitCfg;
    ULONG   ulQBaseAddr;
    ULONG   ulInFreeHeadPtr;
    ULONG   ulInFreeTailPtr;
    ULONG   ulInPostHeadPtr;
    ULONG   ulInPostTailPtr;
    ULONG   ulOutFreeHeadPtr;
    ULONG   ulOutFreeTailPtr;
    ULONG   ulOutPostHeadPtr;
    ULONG   ulOutPostTailPtr;
    ULONG   ulQStatusCtrl;
    UCHAR   uchReserved4[4];
    ULONG   ulPciLocRange1;
    ULONG   ulPciLocRemap1;
    ULONG   ulPciLBRegDescr1;
} CONFIG_REGS, *PCONFIG_REGS;
```

EXAMPLE:

```
HANDLE      hDevice;
DWORD       dwTransferSize;
DWORD       dwErrorCode;
CONFIG_REGS LocalConfigRegs;

if ((! DeviceIoControl(hDevice, IOCTL_HPDI_READ_LOCAL_CONFIG, NULL,
                       0, &LocalConfigRegs, sizeof(CONFIG_REGS),
                       &dwTransferSize, NULL)) ||
    (dwTransferSize != sizeof(CONFIG_REGS)))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
else
{
   printf("\n");
   printf("    LOCAL CONFIGURATION REGISTERS\n");
   printf("Range for PCI to Local 0 Reg    = %08lx\n",
          LocalConfigRegs.ulPciLocRange0);
   printf("Remap for PCI to Local 0 Reg    = %08lx\n",
          LocalConfigRegs.ulPciLocRemap0);
   printf("Mode Arbitration Reg            = %08lx\n",
          LocalConfigRegs.ulModeArb);
   printf("Big/Little Endian Descr. Reg    = %08lx\n",
          LocalConfigRegs.ulEndianDescr);
   printf("Range for PCI to Local Reg      = %08lx\n",
          LocalConfigRegs.ulPciLERomRange);
   printf("Remap for PCI to Local Reg      = %08lx\n",
          LocalConfigRegs.ulPciLERomRemap);
   printf("Bus Region Descriptions for Reg = %08lx\n",
          LocalConfigRegs.ulPciLBRegDescr0);
   printf("Range for Local to PCI Reg      = %08lx\n",
          LocalConfigRegs.ulLocPciRange);
   printf("Base Addr for Local to PCI Reg  = %08lx\n",
          LocalConfigRegs.ulLocPciMemBase);
```

```
printf("Base Addr for Local to PCI Reg  = %08lx\n",
       LocalConfigRegs.ulLocPciIOBase);
printf("Remap for Local to PCI Reg      = %08lx\n",
       LocalConfigRegs.ulLocPciRemap);
printf("PCI Config Address Reg for Reg  = %08lx\n",
       LocalConfigRegs.ulLocPciConfig);
printf("Range for PCI to Local 1 Reg    = %08lx\n",
       LocalConfigRegs.ulPciLocRange1);
printf("Remap for PCI to Local 1 Reg    = %08lx\n",
       LocalConfigRegs.ulPciLocRemap1);
printf("Bus Region Descriptor Reg       = %08lx\n",
       LocalConfigRegs.ulPciLBRegDescr1);

printf("    RUNTIME REGISTERS\n");
printf("Mailbox Register 0          = %08lx\n",
       LocalConfigRegs.ulMailbox[0]);
printf("Mailbox Register 1          = %08lx\n",
       LocalConfigRegs.ulMailbox[1]);
printf("Mailbox Register 2          = %08lx\n",
       LocalConfigRegs.ulMailbox[2]);
printf("Mailbox Register 3          = %08lx\n",
       LocalConfigRegs.ulMailbox[3]);
printf("Mailbox Register 4          = %08lx\n",
       LocalConfigRegs.ulMailbox[4]);
printf("Mailbox Register 5          = %08lx\n",
       LocalConfigRegs.ulMailbox[5]);
printf("Mailbox Register 6          = %08lx\n",
       LocalConfigRegs.ulMailbox[6]);
printf("Mailbox Register 7          = %08lx\n",
       LocalConfigRegs.ulMailbox[7]);
printf("PCI to Local Doorbell Reg   = %08lx\n",
       LocalConfigRegs.ulPciLocDoorBell);
printf("Local to PCI Doorbell Reg   = %08lx\n",
       LocalConfigRegs.ulLocPciDoorBell);
printf("Interrupt Control/Status    = %08lx\n",
       LocalConfigRegs.ulIntCntrlStat);
printf("EEPROM Control, PCI Command = %08lx\n",
       LocalConfigRegs.ulRunTimeCntrl);
printf("Device ID                   = %08lx\n",
       LocalConfigRegs.ulDeviceVendorID);
printf("Revision ID                 = %08lx\n",
       LocalConfigRegs.ulRevisionID);
printf("Mailbox Register 0          = %08lx\n",
       LocalConfigRegs.ulMailboxReg0);
printf("Mailbox Register 1          = %08lx\n",
       LocalConfigRegs.ulMailboxReg1);

printf("    DMA REGISTERS\n");
printf("dma channel 0 mode Reg           = %08lx\n",
       LocalConfigRegs.ulDMAMode0);
printf("dma channel 0 pci address Reg    = %08lx\n",
       LocalConfigRegs.ulDMAPCIAddress0);
printf("dma channel 0 local address Reg  = %08lx\n",
       LocalConfigRegs.ulDMALocalAddress0);
printf("dma channel 0 transfer byte Reg  = %08lx\n",
       LocalConfigRegs.ulDMAByteCount0);
printf("dma channel 0 descriptor Reg     = %08lx\n",
       LocalConfigRegs.ulDMADescriptorPtr0);
printf("dma channel 1 mode Reg           = %08lx\n",
       LocalConfigRegs.ulDMAMode1);
printf("dma channel 1 pci address Reg    = %08lx\n",
```

```
            LocalConfigRegs.ulDMAPCIAddress1);
    printf("dma channel 1 local address Reg  = %08lx\n",
            LocalConfigRegs.ulDMALocalAddress1);
    printf("dma channel 1 transfer byte Reg  = %08lx\n",
            LocalConfigRegs.ulDMAByteCount1);
    printf("dma channel 1 descriptor Reg     = %08lx\n",
            LocalConfigRegs.ulDMADescriptorPtr1);
    printf("dma command/status registers Reg = %08lx\n",
            LocalConfigRegs.ulDMACmdStatus);
    printf("dma arbitration register Reg     = %08lx\n",
            LocalConfigRegs.ulDMAArbitration);
    printf("dma threshold register Reg       = %08lx\n",
            LocalConfigRegs.ulDMAThreshold);

    printf("MESSAGING QUEUE REGISTERS\n");
    printf("outbound post queue Int Status Reg = %08lx\n",
            LocalConfigRegs.ulOutPostQIntStatus);
    printf("outbound post queue Int Mask Reg   = %08lx\n",
            LocalConfigRegs.ulOutPostQIntMask);
    printf("Mailbox Reg 0                      = %08lx\n",
            LocalConfigRegs.ulMailbox[0]);
    printf("Mailbox Reg 1                      = %08lx\n",
            LocalConfigRegs.ulMailbox[1]);
    printf("messaging unit configuration Reg   = %08lx\n",
            LocalConfigRegs.ulMsgUnitCfg);
    printf("queue base address register Reg    = %08lx\n",
            LocalConfigRegs.ulQBaseAddr);
    printf("inbound free head pointer Reg      = %08lx\n",
            LocalConfigRegs.ulInFreeHeadPtr);
    printf("inbound free tail pointer Reg      = %08lx\n",
            LocalConfigRegs.ulInFreeTailPtr);
    printf("inbound post head pointer Reg      = %08lx\n",
            LocalConfigRegs.ulInPostHeadPtr);
    printf("inbound post tail pointer Reg      = %08lx\n",
            LocalConfigRegs.ulInPostTailPtr);
    printf("inbound free head pointer Reg      = %08lx\n",
            LocalConfigRegs.ulOutFreeHeadPtr);
    printf("inbound free tail pointer Reg      = %08lx\n",
            LocalConfigRegs.ulOutFreeTailPtr);
    printf("inbound post head pointer Reg      = %08lx\n",
            LocalConfigRegs.ulOutPostHeadPtr);
    printf("inbound post tail pointer Reg      = %08lx\n",
            LocalConfigRegs.ulOutPostTailPtr);
    printf("queue status/control Reg           = %08lx\n",
            LocalConfigRegs.ulQStatusCtrl);
}
```

4.5.15. IOCTL_HPDI_WRITE_PCI_CONFIG_REG

The IOCTL_HPDI_WRITE_PCI_CONFIG_REG function will write a value to one of the PCI Configuration Registers. The user should be very careful modifying values of certain registers. The following registers should not be changed:

- PCI_MEM_BASE_ADDR
- PCI_IO_BASE_ADDR
- PCI_BASE_ADDR_0
- PCI_BASE_ADDR_1
- PCI_BASE_ADDR_LOC_ROM

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_REGISTER_PARAMS
{
    ULONG eHPDIRegister;
    ULONG ulRegisterValue;
} HPDI_REGISTER_PARAMS, *PHPDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eHPDIRegister is one of the following:

#define STATUS_COMMAND               1
#define BIST_HDR_TYPE_LAT_CACHE_SIZE  3
#define PCI_MEM_BASE_ADDR            4
#define PCI_IO_BASE_ADDR             5
#define PCI_BASE_ADDR_0              6
#define PCI_BASE_ADDR_1              7
#define PCI_BASE_ADDR_LOC_ROM       12
#define LAT_GNT_INT_PIN_LINE        15
```

EXAMPLE:

```
HANDLE                hDevice;
DWORD                 dwTransferSize;
DWORD                 dwErrorCode;
HPDI_REGISTER_PARAMS  InputRegData;

InputRegData.eHPDIRegister   = STATUS_COMMAND;
InputRegData.ulRegisterValue = 0x12345678;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_WRITE_PCI_CONFIG_REG,
                      &InputRegData,  sizeof(HPDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.16. IOCTL_HPDI_WRITE_LOCAL_CONFIG_REG

The IOCTL_HPDI_WRITE_LOCAL_CONFIG_REG function will write a value to one of the Local Configuration Registers. The user should be very careful modifying values of certain registers. All of the DMA Registers should not be changed while a data transfer is in progress. The following registers should not be changed:

- All Local Configuration Registers

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_REGISTER_PARAMS
{
    ULONG eHPDIRegister;
    ULONG ulRegisterValue;
} HPDI_REGISTER_PARAMS, *PHPDI_REGISTER_PARAMS;

Where ulRegisterValue contains the value to be written to the
register and eHPDIRegister is one of the following:

/*** DMA Registers ***/
#define DMA_CH_0_MODE                32
#define DMA_CH_0_PCI_ADDR            33
#define DMA_CH_0_LOCAL_ADDR          34
#define DMA_CH_0_TRANS_BYTE_CNT      35
#define DMA_CH_0_DESC_PTR            36
#define DMA_CH_1_MODE                37
#define DMA_CH_1_PCI_ADDR            38
#define DMA_CH_1_LOCAL_ADDR          39
#define DMA_CH_1_TRANS_BYTE_CNT      40
#define DMA_CH_1_DESC_PTR            41
#define DMA_CMD_STATUS               42
#define DMA_MODE_ARB_REG             43
#define DMA_THRESHOLD_REG            44

/*** Local Configuration Registers. ***/
#define PCI_TO_LOC_ADDR_0_RNG         0
#define LOC_BASE_ADDR_REMAP_0         1
#define MODE_ARBITRATION              2
#define BIG_LITTLE_ENDIAN_DESC        3
#define PCI_TO_LOC_ROM_RNG            4
#define LOC_BASE_ADDR_REMAP_EXP_ROM   5
#define BUS_REG_DESC_0_FOR_PCI_LOC    6
#define DIR_MASTER_TO_PCI_RNG         7
#define LOC_ADDR_FOR_DIR_MASTER_MEM   8
#define LOC_ADDR_FOR_DIR_MASTER_IO    9
#define PCI_ADDR_REMAP_DIR_MASTER    10
#define PCI_CFG_ADDR_DIR_MASTER_IO   11
#define PCI_TO_LOC_ADDR_1_RNG        92
#define LOC_BASE_ADDR_REMAP_1        93
#define BUS_REG_DESC_1_FOR_PCI_LOC   94
```

```
/*** Run Time Registers ***/
#define MAILBOX_REGISTER_0          16
#define MAILBOX_REGISTER_1          17
#define MAILBOX_REGISTER_2          18
#define MAILBOX_REGISTER_3          19
#define MAILBOX_REGISTER_4          20
#define MAILBOX_REGISTER_5          21
#define MAILBOX_REGISTER_6          22
#define MAILBOX_REGISTER_7          23
#define PCI_TO_LOC_DOORBELL         24
#define LOC_TO_PCI_DOORBELL         25
#define INT_CTRL_STATUS             26
#define PROM_CTRL_CMD_CODES_CTRL    27
#define DEVICE_ID_VENDOR_ID         28
#define REVISION_ID                 29
#define MAILBOX_REG_0               30
#define MAILBOX_REG_1               31

/*** Messaging Queue Registers ***/
#define OUT_POST_Q_INT_STATUS       12
#define OUT_POST_Q_INT_MASK         13
#define IN_Q_PORT                   16
#define OUT_Q_PORT                  17
#define MSG_UNIT_CONFIG             48
#define Q_BASE_ADDR                 49
#define IN_FREE_HEAD_PTR            50
#define IN_FREE_TAIL_PTR            51
#define IN_POST_HEAD_PTR            52
#define IN_POST_TAIL_PTR            53
#define OUT_FREE_HEAD_PTR           54
#define OUT_FREE_TAIL_PTR           55
#define OUT_POST_HEAD_PTR           56
#define OUT_POST_TAIL_PTR           57
#define Q_STATUS_CTRL_REG           58
```

EXAMPLE:

```
HANDLE                hDevice;
DWORD                 dwTransferSize;
DWORD                 dwErrorCode;
HPDI_REGISTER_PARAMS  InputRegData;

InputRegData.eHPDIRegister   = MAILBOX_REGISTER_0;
InputRegData.ulRegisterValue = 0x99999999;

if (! DeviceIoControl(hDevice, IOCTL_HPDI_WRITE_LOCAL_CONFIG_REG,
                      &InputRegData,  sizeof(HPDI_REGISTER_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.17. IOCTL_HPDI_SET_READ_TIMEOUT

The IOCTL_HPDI_SET_READ_TIMEOUT function will set the timeout that the driver uses for ending read operations when enough data is not available. The time is specified in seconds. A –1 will indicate no timeout. The default time set when the driver is initialized is 2 seconds.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_SET_TIMEOUT_PARAMS
{
   LONG lTimeout;
} HPDI_SET_TIMEOUT_PARAMS, *PHPDI_SET_TIMEOUT_PARAMS;
```

EXAMPLE:

```
HANDLE                  hDevice;
DWORD                   dwTransferSize;
DWORD                   dwErrorCode;
HPDI_SET_TIMEOUT_PARAMS Timeout;

/* Set the time to never timeout. */
Timeout.lTimeout = -1L;
if (! DeviceIoControl(hDevice, IOCTL_HPDI_SET_READ_TIMEOUT,
                      &Timeout, sizeof(HPDI_SET_TIMEOUT_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.18. IOCTL_HPDI_SET_WRITE_TIMEOUT

The IOCTL_HPDI_SET_WRITE_TIMEOUT function will set the timeout that the driver uses for ending write operations when there is not enough room in the transmit FIFO. The time is specified in seconds. A –1 will indicate no timeout. The default time set when the driver is initialized is 2 seconds.

Input/Output Buffer:

```
<from HPDIIoctl.h>

typedef struct _HPDI_SET_TIMEOUT_PARAMS
{
   LONG lTimeout;
} HPDI_SET_TIMEOUT_PARAMS, *PHPDI_SET_TIMEOUT_PARAMS;
```

EXAMPLE:

```
HANDLE                   hDevice;
DWORD                    dwTransferSize;
DWORD                    dwErrorCode;
HPDI_SET_TIMEOUT_PARAMS Timeout;

/* Set the time to never timeout. */
Timeout.lTimeout = -1L;
if (! DeviceIoControl(hDevice, IOCTL_HPDI_SET_WRITE_TIMEOUT,
                      &Timeout, sizeof(HPDI_SET_TIMEOUT_PARAMS),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.19. IOCTL_HPDI_SET_SET_DMA_ENABLE

The IOCTL_HPDI_SET_DMA_ENABLE function will set the enable for DMA operations. If DMA is enabled the driver will perform DMA reads and writes when read or write operations are requested. If DMA is not enabled the driver will just perform programmed I/O transfers when read or write operations are requested.

Input/Output Buffer:

```
BOOLEAN bEnable;
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
BOOLEAN  bEnable;

/* Enable DMA. */
bEnable = TRUE;
if (! DeviceIoControl(hDevice, IOCTL_HPDI_SET_DMA_ENABLE,
                      &bEnable, sizeof(BOOLEAN),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

4.5.20. IOCTL_HPDI_SET_DEMAND_MODE_ENABLE

The IOCTL_HPDI_SET_DEMAND_MODE_ENABLE function will set the enable for DMA demand mode. Demand mode DMA is used to enable throttling DMA write operations when the FIFO is almost full and throttling DMA read operations when the FIFO is almost empty.

Input/Output Buffer:

```
BOOLEAN bEnable;
```

EXAMPLE:

```
HANDLE   hDevice;
DWORD    dwTransferSize;
DWORD    dwErrorCode;
BOOLEAN  Timeout;

/* Enable demand mode. */
bEnable = TRUE;
if (! DeviceIoControl(hDevice, IOCTL_HPDI_SET_DEMAND_MODE_ENABLE,
                      &bEnable, sizeof(BOOLEAN),
                      NULL, 0, &dwTransferSize, NULL))
{
   dwErrorCode = GetLastError();
   ErrorMessage("DeviceIoControl", dwErrorCode);
}
```

5. Driver Installation

This section will describe the procedure for installing the PCI-HPDI-32 Windows NT Driver. The following is the installation procedure:

- Insert the installation floppy disk into a 3 ½" floppy drive
- Click on Run from the Start menu
- Type in "**A:\Setup.exe**" and Click the OK button in the Run Dialog Box
- Follow the instructions on the screen
- Either allow the install program to reboot the computer or reboot it manually so the driver will automatically be installed

The following files are installed in the selected directory by the install program:

- **HPDIDriver.sys** – a copy of the driver file installed in the O/S **drivers** directory
- **HPDIIoctl.h** – the 'C' header file that contains the driver access constants and structures. This file should be **#include**'d in application code where the driver is accessed.
- **HPDITest.c** – a 'C' source file containing an example program that shows how to access each of the driver entry points
- **HPDITest.exe** – compiled version of **HPDITest.c** that will allow menu access to each of driver entry points
- **readme.txt** – a file containing the latest information on the driver
- **Uninst.isu** – a file containing information that allows the driver to be uninstalled

The driver is installed to be automatically started up when the computer is booted.

6. Test Program

This section will describe how to execute the test program installed with the PCI-HPDI-32 driver. The following is the procedure for executing the test program:

- Start up a command prompt window
- Change to the directory where the driver was installed
- Type "**HPDITest \\.\hpdix**", where x is the number of the PCI-HPDI-32 board to access, starting with 1 for the first board