

PCIHSD® Interface Board
PCIHSD2® Interface Board

Software Device Driver
Windows NT 4.0/Windows 2000
User Manual

Document Number: 0900120
Rev. 1.1

APPLIED DATA SCIENCES, INC. has contained in this subject matter proprietary material and all manufacturing, reproduction, use and sales rights pertaining to this material are expressly reserved. This material is submitted in confidence for a specified purpose and the user, by accepting this material, agrees that it will not be used, copied or reproduced in whole or in part except to meet the purpose for which it was provided.

APPLIED DATA SCIENCES, INC. reserves the right to make design changes or modification to any product to improve performance or incorporate new functions. The material in this document is for informational purposes and is subject to change without notice.

APPLIED DATA SCIENCES, INC. assumes no responsibility for any errors that may appear in this document.

TO CONTACT US

Our mailing address is:

Applied Data Sciences, Inc.
P.O. Box 814209
Dallas, TX 75381-4209
USA

Our shipping address is:

Applied Data Sciences, Inc.
1300 North I-35E, Suite 100
Carrollton, TX 75006
USA

Our telephone numbers are:

972-242-7944 (USA & international)

Our facsimile number is:

972-242-8874 (USA & international)

Our email address is:

support@appdatasci.com

Copyright © 2000 by
APPLIED DATA SCIENCES, INC.
All Rights Reserved.
Printed in the U.S.A.

REVISION STATUS

<u>REV</u>	<u>PAGES CHANGED</u>	<u>DESCRIPTION</u>	<u>APPR</u>	<u>DATE</u>
1.0	----	Initial Release	RLT	06/30/2000
1.1	vi,1-2,2-1,3-2,4-1,5-1 through 5-3	Update to include Win2000	TFC	08/06/2001

LIST OF RELATED DOCUMENTS

<u>TITLE</u>	<u>DOCUMENT NUMBER</u>
Encore CSD	
HSDII Technical Manual	303-000270-200
HSDII Hardware Reference Manual	301-320050-000
MPX-32 Reference Manual, Volume I	323-001011-300
Applied Data Sciences	
PCIHSD Interface Board - Technical Manual	0900098
PCIHSD2 Interface Board – Technical Manual	0900117

TRADEMARK ACKNOWLEDGEMENTS

PCIHSD[®] is a registered trademark of Applied Data Sciences.

PCIHSD2[®] is a registered trademark of Applied Data Sciences.

Windows NT[®] is a registered trademark of Microsoft Corporation.

Windows 2000[®] is a registered trademark of Microsoft Corporation.

TABLE OF CONTENTS

SECTION	TITLE	PAGE
1.0	INTRODUCTION	
1.1	PCIHSD Board Description	1-1
1.2	Encore HSDII Board Description/Emulation	1-2
1.3	Software Overview	1-2
1.4	Requirements	1-2
2.0	DRIVER FUNCTIONS	
2.1	General Information	2-1
2.2	PCIHSD Devices in Windows NT/2000	2-1
2.3	IBL Mode Support	2-2
2.4	Symbol Definitions	2-2
2.5	CreateDevice Function	2-2
2.6	CloseHandle Function	2-3
2.7	DeviceIoControl Function	2-3
2.7.1	Set PCIHSD Operating Mode	2-3
2.7.2	Perform PCIHSD Reset Operations	2-3
2.7.3	Get PCIHSD Board Status	2-4
2.7.4	Start PCIHSD Input/Output	2-4
2.7.5	Terminate PCIHSD Input/Output	2-5
2.7.6	Set PCIHSD Slow Timeout	2-5
2.7.7	Set PCIHSD Fast Timeout	2-5
2.7.8	PCIHSD Timeout Control	2-6
3.0	DATA STRUCTURES	
3.1	Introduction	3-1
3.2	Reference Documents	3-1
3.3	Set Mode Variable	3-1
3.4	PCIHSD_STATUS Structure	3-2
3.5	PCIHSD_TIMECTRL Structure	3-3
4.0	STATUS RETURNS	
4.1	Status Returns	4-1

5.0 PROGRAM INSTALLATION

5.1	Distribution Media	5-1
5.2	Installation	5-1
5.3	Completing the Installation	5-2
5.4	Driver Loading	5-2

APPENDICES

<u>APPENDIX</u>	<u>TITLE</u>	<u>PAGE</u>
A	Example Program	A-1

SECTION 1.0 INTRODUCTION

1.1 PCIHSD Board Description

The PCIHSD® or PCIHSD2® board provides a high-speed bi-directional link for transferring control, status, and data between a 32-bit Peripheral Component Interconnect (PCI) bus compatible system, and the Encore High-Speed Digital Interface (HSDII) or any 32-bit external device using the Encore High-Speed Digital Interface (HSDII) protocol.

The PCIHSD2 is a half-length PCI board residing in a 32-bit 5-volt PCI slot in the host computer chassis. There is one (1) 100-pin connector on the rear edge of the PCIHSD2 board. This 100-pin connector attaches to a mating 100-pin connector (with latches) which in turn has crimped to it (2) 50-pin flat ribbon cables. These two flat ribbon cables have 50-pin IDC female connectors at their opposite end. Using the supplied 50-pin flat ribbon cable harness, connection is made between the PCIHSD2 board, and the HSDII card or HSD compatible device.

All power for the PCIHSD2 board is supplied by the internal host computer power supply.

There are no hardware switches. All setup and control is handled via software. The PCIHSD can be configured for the following modes:

- HSD,
- IBL or
- Device EM (External Mode).

In the HSD mode the PCIHSD board emulates the Encore HSDII board. In the IBL mode the PCIHSD board can be attached directly to an Encore HSDII board or to another PCIHSD board. In the Device External Mode the PCIHSD board emulates the device driving the Encore HSDII board in External Mode.

This high-speed data link provides a 32-bit data bus transfer between the host computer memory and the PCIHSD board. The data are 32-bit word transfers between the PCIHSD board and HSDII compatible external device.

For additional technical specifications of this board refer to the PCIHSD2 Technical Manual, part number 0900117.

This software can also be used with the older version PCIHSD, part number 0700520, which is electrically equivalent to the PCIHSD. In the following sections, the term PCIHSD refers equally to the PCIHSD and PCIHSD2.

1.2 Encore HSDII Board Description/Emulation

This information is presented to help you understand the purpose of the PCIHSD board - that is to emulate the functions of the Encore HSDII board. The HSDII board is a high-speed general-purpose interface board, which is installed in an Encore computer chassis. This board is manufactured by Encore, Inc. Computer Systems Division, Fort Lauderdale, Florida.

The Encore HSDII board provides a full 32-bit parallel interface to an external device which operates autonomously under control of an Input/Output Control List (IOCL). A detailed description of the HSDII board is found in the HSDII Technical Manual and HSDII Hardware Reference Manual. The document part numbers of these two manuals are contained in the index under LIST OF RELATED DOCUMENTS.

The PCIHSD also provides a 32-bit parallel interface to an external device with the capability of both Programmed I/O (PIO) and Direct Memory Access (DMA) data transfers. Some functions performed by the Encore HSDII must be emulated by host software in the context of the PCIHSD. Once the host has initiated a DMA data transfer, the external device executes transfers between the PCIHSD's onboard FIFO and the external device independent of the host's operation. The handshake seen by the external device appears functionally identical to the Encore HSDII card.

1.3 Software Overview

This document provides all the information necessary to integrate the user's software with the PCIHSD.

The PCIHSD driver, "**PCIHSD.sys**", is intended for operation under Windows NT Version 4.0. The driver handles all communication with the PCIHSD on behalf of the calling program. It allows for reading and setting the operating mode of the PCIHSD.

Details of the driver installation are in Section 5.0.

1.4 Requirements

Software:	Window NT, Version 4.0, or Windows 2000
Hardware:	x86 Personal Computer with PCI bus. PCIHSD Card.

Knowledge of the Encore HSDII IOCB structure will aid in using the PCIHSD board and software.

SECTION 2.0 DRIVER FUNCTIONS

2.1 General Information

The PCIHSD driver is installed as a standard Windows NT/2000 device driver. It can handle any number of PCIHSD boards configured for either HSD or IBL mode. The following system calls are supported:

CreateFile
CloseHandle
DeviceIoControl

All access to the PCIHSD device is made through **DeviceIoControl** calls. Arguments to the **DeviceIoControl** call provide for reading and writing data, for setting the device operation mode, for reading status, for resetting the PCIHSD, and for varying device timeout on input/output operations.

2.2 PCIHSD Devices in Windows NT/2000

The PCIHSD driver is registered in Windows NT/2000 as both a device and an event-logging driver. The driver can service any number of PCIHSD devices and can operate in a multi-processor environment.

The driver logs events at the time it loads or unloads. Any error that occurs during loading that prevents the driver from remaining loaded is also logged. These events can be viewed using the Event Viewer provided with Windows NT/2000.

The device name for a PCIHSD device is as follows:

\\.\PCIHSDn

where 'n' is a digit starting with 0 (zero) for the first device and incrementing by one for each additional device in the system.

2.3 IBL Mode Support

The PCIHSD driver provides three protocols for initiating a link for data transfers in the InterBus Link (IBL) configuration. The desired mode is specified in the variable passed to the **DeviceIoControl** system call to **Set PCIHSD Operating Mode**. If **PCIHM_LREQ** is specified, the PCIHSD will initiate the link request for all transfers. If **PCIHM_LACK** is specified, the PCIHSD does not initiate any link request, but expects to wait for, and acknowledge a link request before each data transfer. A third option uses the protocol of the Encore H.IBLG handler under MPX and initiates the link request when the operation is a Write or waits for and acknowledges a link request when the operation is a Read. This method is specified by **PCIHM_LIBLG**.

2.4 Symbol Definitions

The header files, **pcihsd.h** and **pcihsdioctl.h**, contain the definitions of symbols used with the **DeviceIoControl** system call to perform various HSD functions, as well as declarations for data structures used with those calls.

2.5 CreateDevice Function

The standard **CreateDevice** function is called to associate a device handle with the desired PCIHSD device. For example, the most likely parameters to create the PCIHSD device is:

```
TCHAR      csDeviceName[] = "\\.\PCIHSD0";
DWORD     dwAccess = GENERIC_READ|GENERIC_WRITE;
DWORD     dwShareMode = 0;
DWORD     dwCreationDisp = OPEN_EXISTING;
DWORD     dwFlags = FILE_ATTRIBUTE_NORMAL;
HANDLE    hPciHsd;

hPciHsd = CreateFile( csDeviceName,
                    dwAccess,
                    dwShareMode,
                    NULL,           // security attributes
                    dwCreationDisp,
                    dwFlags,
                    NULL           // template file handle
                    );
```

It is possible to perform asynchronous operations on the PCIHSD device by specifying the **FILE_FLAG_OVERLAPPED** flag as part of the **dwFlags** parameter.

When the device is created, the PCIHSD is reset and the driver sets a five-second timeout default for all I/O operations.

2.6 CloseHandle Function

The standard **CloseHandle** function must be called to terminate access to the PCIHSD device. Any pending operations for the open device are cancelled and the associated memory areas released when the device is closed.

2.7 DeviceIoControl Function

The **DeviceIoControl** call is used to perform all operations on the PCIHSD Device. The call requires the user to pass the file handle assigned to the device, a function code and input/output data specific to the function requested. The function may return a value of 0 (zero) indicating an error. The specific error may be retrieved with the function **GetLastError** and interpreted using the function **FormatMessage**.

The function codes are defined in the **pcihsdioctl.h** include file and special parameters are defined in the **pcihsd.h** include file.

2.7.1 Set PCIHSD Operating Mode

Function Code	IOCTL_HSD_SET_MODE
Input Buffer:	pointer to ULONG variable containing the new settings
Output Buffer:	pointer to ULONG variable that receives the new configuration
Output Length:	size of ULONG variable.

This function sets the current operating mode of the addressed PCIHSD from information found in the variable pointed to by the Input Buffer parameter. The data that can appear in the variable is defined in Section 3. If a value is not specified for a particular group, the current setting is retained. Finally, the new operating mode information is returned to the variable pointed to by the Output Buffer parameter.

2.7.2 Perform PCIHSD Reset Operations

Function Code:	IOCTL_HSD_RESET
Input Buffer:	pointer to ULONG variable specifying the type of reset
Input Length:	size of ULONG variable

This call performs reset operations on the addressed PCIHSD. The variable pointed to by the Input Buffer parameter specifies the type of reset:

RESET_HSD	Reset HSD controller
RESET_FIFO	Reset FIFO's
RESET_BOTH	Reset FIFO's and HSD controller
RESET_DVC	Reset external device (IOReset)
RESET_TDV	Issue Terminate Device Signal to External Device
RESET_GEN	General reset (HSD/FIFO/External Device)

2.7.3 Get PCIHSD Board Status

Function Code:	IOCTL_HSD_GET_STATUS
Output Buffer:	pointer to <i>PCIHSD_STATUS</i> structure
Output Length:	size of <i>PCIHSD_STATUS</i> structure

This function stores into the structure, pointed to by the Output Buffer parameter, the current board status word and HSD sequencer state word.

2.7.4 Start PCIHSD Input/Output

Function Code:	IOCTL_HSD_START
Input Buffer:	pointer to an Encore type IOCB list array
Input Length:	size of the input array
Output Buffer:	pointer to an array that will receive the IOCB list after completed
Output Length:	size of the output array

This function is used to present a list of IOCBs that perform the operation desired by the user. The user must be familiar with the Encore IOCB design and IOCB fields. The IOCB structure is defined in the include file, **pcihsdiocctl.h**.

The IOCB list can be made up of all the commands permitted by the HSD except for a Transfer In Channel (TIC) command. The IOCBs are linked together by either the Command Chain bit or the Data Chain bit as required. The driver processes the entire list, transforming the IOCBs into appropriate PCIHSD descriptors to execute the operation. Since data is moved directly to or from the user's data arrays, those areas of the user's memory are locked into physical memory for the duration of the call.

The Output Array is optional. The only time it is necessary is for those IOCB lists that contain a Device Status Request IOCB. In this case, the device status and residual count is stored in the fourth word of the IOCB. The user must provide an area for the system to return the list to get the device status information.

2.7.5 Terminate PCIHSD Input/Output

Function Code: **IOCTL_HSD_TERMINATE**

This function causes any current Input/Output operation to cease.

2.7.6 Set PCIHSD Slow Timeout

Function Code: **IOCTL_HSD_SET_TIMEOUT**
Input Buffer: pointer to LONG variable containing the interval timeout value
Input Length: size of LONG variable
Output Buffer: pointer to LONG variable receiving the previous timeout value
Output Length: size of LONG variable

This function assumes the value pointed to by the Input Buffer is in units of seconds and is capable of specifying a long timeout on data transfers through the PCIHSD.

Note: a fast timeout value takes precedence over a slow timeout value. The fast timeout value must be reset to zero for the slow timeout value to be used.

2.7.7 Set PCIHSD Fast Timeout

Function Code: **IOCTL_HSD_SET_FAST_TIMEOUT**
Input Buffer: pointer to LONG variable containing the interval timeout value
Input Length: size of LONG variable
Output Buffer: pointer to LONG variable receiving the previous timeout value
Output Length: size of LONG variable

This function assumes the value pointed to by the Input Buffer is in units of milliseconds and is capable of specifying a short timeout on data transfers through the PCIHSD. To cancel a fast timeout, the value of the fast timeout must be set to zero. See note above.

2.7.8 PCIHSD Timeout Control

Function Code:	IOCTL_HSD_TIMEOUT_CONTROL
Input Buffer:	pointer to a PCIHSD_TIMECTRL structure with new values
Input Length:	size of PCIHSD_TIMECTRL variable
Output Buffer:	pointer to PCIHSD_TIMECTRL structure for the old values
Output Length:	size of PCIHSD_TIMECTRL variable

This function is the preferred method of setting timeout values to govern Input/Output operations. With this function, the timeout method used by the driver is set to correspond to the units of the time value. If the time value is in milliseconds, then the driver employs a high-speed timer and timeouts can be as short as 20 milliseconds. If the time value is in seconds, then the driver employs a low-speed timer with the minimum timeout being one second. See the description of the PCIHSD_TIMECTRL structure in Section 3 for information on the contents of the structure.

SECTION 3.0 DATA STRUCTURES

3.1 Introduction

This section describes the data structures and variables used with the **DeviceIoControl** function calls to perform I/O operations with the PCIHSD. Data structures and data constants are defined in the file **pcihsd.h**.

3.2 Reference Documents

Refer to the respective Encore MPX-32 Operating System Reference and Technical software manuals for a detailed description of the software requirements for the IOCB structure and the HSDII board operation. For a comprehensive hardware description of the Encore HSDII compatible card refer to:

Encore HSDII Technical Manual, Document # 303-329131-000
Encore MPX-32 Volume 2, Reference Manual, Document #323-001011-300

Refer to the following documents for assistance programming the PCIHSD card:

ADS PCIHSD Technical Manual, Document # 0900098
ADS PCIHSD2 Technical Manual, Document # 0900117

3.3 Set Mode Variable

The variable passed in the Set Mode **DeviceIoControl** function contains the following groups of items:

Op Mode: Board operating mode code containing one of the following defined values:

PCIM_HSD	operating as HSD
PCIM_IBL	operating as IBL

Connect: Board connector configuration code containing one of the following defined values:

PCIM_CNRM	Normal (HSD) connectors
PCIM_CREV	Reverse (IBL) connectors

Note: on an IBL link, certain signals must be transposed from one HSD device to the other. This is accomplished by either 1) using a cable with the appropriate wires interchanged and normal connectors on both HSD's or 2) using a straight-through cable and setting one of the HSD devices in a Reverse connector configuration.

Byte Swap: Data swap option containing one of the following defined values:

PCIM_BSWAP	Swap bytes/words
PCIM_NSWAP	No swap

Note: byte swapping is dependent on the type of processor running Windows NT/2000. For those systems running on an Intel x86 processor, select PCIM_NSWAP for this option.

Link type: IBL link request mode containing a code specifying the desired IBL link protocol.

PCIM_LREQ	Always request IBL link when initiating a transfer.
PCIM_LACK	Never request IBL link when initiating transfer.
PCIM_LIBLG	Use link protocol of the Encore H.IBLG handler for MPX. (Request link if writing, don't request if reading.)

Link prior: IBL link request priority containing a code specifying the hardware link priority desired.

PCIM_HIPRI	Set high link priority.
PCIM_LOPRI	Set low link priority.

3.4 PCIHSD_STATUS Structure

The *PCIHSD_STATUS* structure is used with the Get Status **DeviceIoControl** call to return PCIHSD status. This structure contains the following information (all are unsigned 32-bit words):

ulBoardStatus	Current PCIHSD status.
ulState	Current HSD emulation sequencer state word.

3.5 PCIHSD_TIMECTRL Structure

The *PCIHSD_TIMECTRL* structure is used with the Time Control **DeviceIoControl** call to set time out values. This structure contains the following information:

- lTimeValue* A signed long integer that holds the desired time out value.

- bTimeValueInMSecs* A boolean value that when set *TRUE*, *lTimeValue* is interpreted in milliseconds. When set *FALSE*, *lTimeValue* is interpreted in seconds.

SECTION 4.0 STATUS RETURNS

4.1 Status Returns

Status returns from the system calls used to access the PCIHSD driver are consistent with other Windows NT/2000 device drivers.

In case of an error, the **DeviceIoControl** system function called will return 0 (zero) and the function **GetLastError** can be called to get the error code. The error codes returned are standard Windows NT/2000 error codes and can be translated into messages using the **FormatMessage** function.

SECTION 5.0 PROGRAM INSTALLATION

5.1 Distribution Media

The PCIHSD device driver is distributed on 3½ inch floppy and contains the following components:

Device driver code:	Part Number 0950110
Test HSD example code	

The following files should be found in the **pcihsd** folder.

PCIHSD.sys	PCIHSD Device Driver.
pcihsd.reg	PCIHSD registration file containing the proper entries for the Registry.
install.bat	Batch file for completing NT 4.0 installation.
Pcihsd.inf	Driver information file for Windows 2000.
pcihsd.h	Include file that defines structures and constants for programming.
pcihsdioctl.h	Include file that defines DeviceIoControl function calls and IOCB structure for programming.

The files found in the **testhsd** folder comprise a complete C++ project which when compiled creates a program for testing the PCIHSD using Applied Data Sciences' HSD Analyzer/Tester. The project is included as an example of PCIHSD programming.

5.2 Windows NT 4.0 Installation

5.2.1 Installing the Driver

Log on Windows NT as "Administrator" or as any user that has access rights for modifying the NT Registry.

Start Windows Explorer and locate the **pcihsd** directory on the diskette. Double click on the **install.bat** file to install the PCIHSD Device Driver. This action causes,

- 1) the **PCIHSD.sys** file to be copied to the **\winnt\system32\drivers** folder.
- 2) **regedit** is executed specifying **pcihsd.reg** as a parameter to properly register the driver with Windows NT and inform Windows NT that the driver is an event logging process.

Reboot the computer.

5.2.2 Driver Loading

The installation process registers the driver with Windows NT as a manual loading device driver. This is intended as a precaution to ensure that the driver will load properly and not cause the Windows NT system any undue grief. Assuming the PCIHSD board has been physically installed in the desired computer system, load the driver through the following steps.

- 1) From the Windows NT **Start** menu, select **Settings**.
- 2) From **Settings**, select **Control Panel**.
- 3) From **Control Panel**, select **Devices**.
- 4) In the **Devices** window, scroll down to the PCIHSD entry. The PCIHSD entry should show no current Status and Startup should be set to Manual.
- 5) Select the PCIHSD entry and click the **Start** button. The system will now attempt to start the PCIHSD device driver. If it starts properly, the Status for the entry will change to Started. If it does not start properly, use the **Event Viewer** to examine the Event Log for the reason why the driver failed to load. Correct the problem and retry starting the driver.
- 6) Once the driver loads properly, in the **Devices** window select the PCIHSD entry and click the **Startup** button. Select the type of start up desired (most likely **Automatic**) since the driver is not required for proper system operation.

5.3 Windows 2000 Installation

On a Windows 2000 system, first install the PCIHSD board in the computer, then boot the system. Windows 2000 will detect a new device on the PCI bus and start the new device wizard. Alternatively, use the Control Panel Install/Remove hardware option to start the wizard. Within the New Device Wizard, perform the following steps:

- 1) Select Search for Suitable driver
- 2) Insert the driver diskette in the floppy drive and make sure the Floppy Disk option is checked in the list of places to search.
- 3) The wizard should locate the driver on the disk, click on OK to Install
- 4) The wizard will copy the driver file to the appropriate place and make the required registry entries.

When the installation has completed, re-boot the machine if the Wizard suggests doing so.


```

cCableConnect = 'N';
if( *argv[2] == 'R' )
    cCableConnect = 'R';

if( toupper(*argv[1]) == 'R' )
{
    printf("Testing Read IBL Mode of PCIHSD Board\n");
    TestIBLMode( *argv[1], cCableConnect );
}
else if( toupper(*argv[1]) == 'W' )
{
    printf("Testing Write IBL Mode of PCIHSD Board\n");
    TestIBLMode( *argv[1], cCableConnect );
}
else if( toupper(*argv[1]) == 'H' )
{
    printf("Testing HSD Mode of PCIHSD Board\n");
    TestHSDMode(*argv[1]);
}
else
{
    printf( " ???  invalid command\n\n" );
    usage();
}
}
else
{
    usage();
}

ExitProcess( ERROR_SUCCESS );
}

```

```

static VOID TestHSDMode(char read_write)
{
    HANDLE hDevice;
    DWORD dwErrorCode;
    DWORD dwBytesReturned, dwLength, dwResetType;
    ULONG iMode, iCurConfig;
    LONG iNewTimeout, iOldTimeout;
    DWORD dwBuffer[BFSIZE], dwBuff2[BFSIZE];
    PCIHSD_STATUS sStatus;

    int jk, index;
    int count = 0;
    IOCB iocb[NUM_IOCBS], bcoi[NUM_IOCBS];

    for( jk = 0; jk<NUM_IOCBS; jk++)
    {
        bcoi[jk].w1.w = iocb[jk].w1.w = -1;
        bcoi[jk].w2.w = iocb[jk].w2.w = -1;
        bcoi[jk].w3.w = iocb[jk].w3.w = -1;
        bcoi[jk].w4.w = iocb[jk].w4.w = -1;
    }

    // Open the device
    hDevice = CreateFile(
                                cDeviceName,
                                GENERIC_WRITE|GENERIC_READ,
                                0,
                                NULL,
                                OPEN_EXISTING,
                                FILE_ATTRIBUTE_NORMAL,
                                NULL );
    if( hDevice == INVALID_HANDLE_VALUE )

```

```

    {
        dwErrorCode = GetLastError();
        ErrorMessage( "CreateFile", dwErrorCode );
        ExitProcess( dwErrorCode );
    }

    if( !DeviceIoControl(hDevice, IOCTL_HSD_GET_STATUS,           //Status command
                        NULL, 0,                                //Input Buffer, size
                        &sStatus, sizeof(PCIHSD_STATUS),        //Output Buffer, size
                        &dwBytesReturned,                      //Bytes returned
                        NULL) )                                 //No overlapped IO
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "GetStatus", dwErrorCode);
        ExitProcess( dwErrorCode );
    }

    printf( " BoardStatus: %8.8x\n", sStatus.ulBoardStatus);
    printf( "Configuration: %8.8x\n", sStatus.ulState);

    iNewTimeout = 50;           // set to timeout in milliseconds

ResetTimeout:

    if( !DeviceIoControl(hDevice, IOCTL_HSD_SET_FAST_TIMEOUT, //Fast timeout command
                        &iNewTimeout, sizeof(iNewTimeout),    //Input Buffer, size
                        &iOldTimeout, sizeof(iOldTimeout),     //Output Buffer, size
                        &dwBytesReturned,                      //Bytes returned
                        NULL ) )                                 //No overlapped IO
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "DeviceIoControl", dwErrorCode );
        CloseHandle( hDevice );
        ExitProcess( dwErrorCode );
    }

    printf( "Old Timeout was %ld seconds\n", iOldTimeout);

    // set mode of PciHsd board
    iMode = PCIM_HSD;           // set HSD mode
    iMode |= PCIM_CNRM;        // add in Normal cable connections
    iMode |= PCIM_NSWAP;       // add in No byte swap

    if( !DeviceIoControl(hDevice, IOCTL_HSD_SET_MODE,          //Mode command
                        &iMode, sizeof(iMode),                //Input Buffer
                        &iCurConfig, sizeof(iCurConfig),     //Output Buffer
                        &dwBytesReturned,                      //Bytes returned
                        NULL ) )                                 //No overlapped IO
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "DeviceIoControl", dwErrorCode );
        CloseHandle( hDevice );
        ExitProcess( dwErrorCode );
    }

    printf( "iMode:          %8x\n", iMode);
    printf( "iCurConfig: %8x\n", iCurConfig);

    dwResetType = RESET_BOTH; // = 0; defaults to RESET_BOTH

    if( !DeviceIoControl(hDevice, IOCTL_HSD_RESET,            //Reset command
                        &dwResetType, sizeof(dwResetType),  //Input Buffer, size
                        NULL, 0,                               //Output Buffer, size
                        &dwBytesReturned,                    //Bytes returned
                        NULL) )
    {
        dwErrorCode = GetLastError();
    }

```

```

        ErrorMessage( "DeviceIoControl", dwErrorCode );
        CloseHandle( hDevice );
        ExitProcess( dwErrorCode );
    }

printf("Device Reset - dwResetType: %d\n",dwResetType);

index = 0;

// create IOCB list
iocb[index].w1.f.hsd_cmd = IOc_CMD | IOc_CC; //device command with command chain
iocb[index].w1.f.udd_cmd = 0x01;           //set user byte
iocb[index].w1.f.count = 0;
iocb[index].w2.w = 0x87654321;
iocb[index].w3.w = 0;
iocb[index].w4.w = 0;
index++;

iocb[index].w1.f.hsd_cmd = IOc_RD | IOc_CC; // read data with command chain
iocb[index].w1.f.udd_cmd = 0x01;
iocb[index].w1.f.count = BSIZ;
iocb[index].w2.a = &dwBuffer[0];
iocb[index].w3.w = 0;
iocb[index].w4.w = 0;
index++;

iocb[index].w1.f.hsd_cmd = IOc_RD | IOc_CC; // read data with command chain
iocb[index].w1.f.udd_cmd = 0x01;
iocb[index].w1.f.count = BSIZ;
iocb[index].w2.a = &dwBuff2[0];
iocb[index].w3.w = 0;
iocb[index].w4.w = 0;
index++;

iocb[index].w1.f.hsd_cmd = IOc_DSR;           // device status request
iocb[index].w1.f.udd_cmd = 0x01;
iocb[index].w1.f.count = 0;
iocb[index].w2.w = 0;
iocb[index].w3.w = 0;
iocb[index].w4.w = 0x5a5a5a5a;
index++;

// send IOCB List
dwLength = sizeof( IOCB ) * index;

do
{
    // send the IOCB list for execution
    if( !DeviceIoControl(hDevice, IOCTL_HSD_START,           //IO command
                        &iocb[0], dwLength,               //Input Buffer, size
                        &bcoi[0], dwLength,               //Output Buffer, size
                        &dwBytesReturned,
                        NULL ))
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "DeviceIoControl", dwErrorCode );

        // check on fast timeout value
        if( iNewTimeout )
        {
            // if still on fast timeout, reset it, and try again
            iNewTimeout = 0;
            goto ResetTimeout;
        }

        // close the device and exit
        CloseHandle( hDevice );
        ExitProcess( dwErrorCode );
    }
}

```

```

        // if lower case, check for exit command
        if( read_write == 'h')
        {
            printf( "%d\t\tcycles\r", ++count);
            if( _kbhit() )
            {
                read_write = 'H';
                _getch();
                printf("\n");
            }
        }
        // loop as long as read_write code is lower case H
    }
    while( read_write == 'h' );

    // dump buffers
    for( jk=0; jk<BSIZE; jk++)
    {
        printf("%3d: %8.8x %8.8x\n",jk, dwBuffer[jk],dwBuff2[jk]);
    }

    printf( "Success !\n" );

    printf( "device status = %4.4x\n", bcoi[index-1].w4.f.sts);
    printf( "device count = %4.4x\n", bcoi[index-1].w4.f.cnt);
    printf( "dwLength: %d dwBytesReturned: %d index: %d\n",
            dwLength, dwBytesReturned, index);

    // dump original IOCB list
    printf("\nIOCB List original\n");
    for( jk=0; jk<index; jk++)
    {
        dump_iocb( jk, &iocb[jk]);
    }

    // dump returned IOCB list
    printf("\nIOCB List returned\n");
    for( jk=0; jk<index; jk++)
    {
        dump_iocb( jk, &bcoi[jk]);
    }

    // Close the device
    CloseHandle( hDevice );
}

static VOID dump_iocb( int i, IOCB *iocb)
{
    printf("%3d: %8.8x",i, iocb->w1.w);
    printf(" %8.8x", iocb->w2.w);
    printf(" %8.8x", iocb->w3.w);
    printf(" %8.8x\n", iocb->w4.w);
}

static VOID TestIBLMode(char read_write, char cable_connect)
{
    HANDLE hDevice;
    DWORD dwErrorCode;
    DWORD dwBytesReturned, dwLength;
    ULONG iMode, iCurConfig;
    DWORD dwBuffer[BSIZE];

    int jk, index;
    int count=0;
    IOCB iocb[NUM_IOCBS],bcoi[NUM_IOCBS];

    // fill buffer in case of write
    for( jk=0; jk<BSIZE; jk++)
        dwBuffer[jk] = 0x12345600 +jk;
}

```

```

// Open the device
hDevice = CreateFile(
    cDeviceName,
    GENERIC_WRITE|GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL );
if( hDevice == INVALID_HANDLE_VALUE )
{
    dwErrorCode = GetLastError();
    ErrorMessage( "CreateFile", dwErrorCode );
    ExitProcess( dwErrorCode );
}

// set mode of PciHsd board
iMode = PCIM_IBL; // set IBL mode

if( cable_connect == 'R' )
    iMode |= PCIM_CREV; // add in Reverse cable connections
else
    iMode |= PCIM_CNRM; // add in Normal cable connections

iMode |= PCIM_NSWAP; // add in No byte swap

iMode |= PCIM_LREQ; // set for link request always for HSD Analyzer/Tester
// iMode |= PCIM_LIBLG; // set for link based on read/ write code

if( !DeviceIoControl(hDevice, IOCTL_HSD_SET_MODE,
    &iMode, sizeof(iMode),
    &iCurConfig, sizeof(iCurConfig),
    &dwBytesReturned,
    NULL ))
{
    dwErrorCode = GetLastError();
    ErrorMessage( "DeviceIoControl", dwErrorCode );
    CloseHandle( hDevice );
    ExitProcess( dwErrorCode );
}

printf( "iMode: %8x\n", iMode);
printf( "iCurConfig: %8x\n", iCurConfig);

// set up an IBL IOCB for either Read or Write
index = 0;

iocb[index].w1.f.hsd_cmd = (toupper(read_write) == 'R') ? IOC_RD: IOC_WT;
iocb[index].w1.f.udd_cmd = 0x01;
iocb[index].w1.f.count = BSIZE;
iocb[index].w2.a = &dwBuffer[0];
iocb[index].w3.w = 0;
iocb[index].w4.w = 0;
index++;

// send IOCB List
dwLength = sizeof( IOCB ) * index;

do
{
    if( !DeviceIoControl(hDevice, IOCTL_HSD_START,
        &iocb[0], dwLength,
        &bcoi[0], dwLength,
        &dwBytesReturned,
        NULL ))
    {
        dwErrorCode = GetLastError();
        ErrorMessage( "DeviceIoControl", dwErrorCode );
        CloseHandle( hDevice );
    }
}

```

```

        ExitProcess( dwErrorCode );
    }
    if( (read_write == 'r') || (read_write == 'w') )
    {
        printf(" %d\t\tcycles\r", ++count);
        if( _kbhit() )
        {
            _getch();
            read_write = toupper( read_write );
            printf("\n");
        }
    }
}
while( (read_write == 'r') || (read_write == 'w') );

for( jk=0; jk<BSIZE; jk++)
{
    printf("%3d: %8.8x\n", jk, dwBuffer[jk]);
}

printf( "Success !\n" );

printf("\nIOCB List original\n");
for( jk=0; jk<index; jk++)
{
    printf("%3d: %8.8x", jk, iocb[jk].w1.w);
    printf("   %8.8x", iocb[jk].w2.w);
    printf("   %8.8x", iocb[jk].w3.w);
    printf("   %8.8x\n", iocb[jk].w4.w);
}

printf("\nIOCB List returned\n");
for( jk=0; jk<index; jk++)
{
    printf("%3d: %8.8x", jk, bcoi[jk].w1.w);
    printf("   %8.8x", bcoi[jk].w2.w);
    printf("   %8.8x", bcoi[jk].w3.w);
    printf("   %8.8x\n", bcoi[jk].w4.w);
}

// Close the device
CloseHandle( hDevice );
}

//
// This helper routine converts a system-service error
// code into a text message and prints it on StdOutput
//
static VOID
ErrorMessage(
    LPTSTR lpOrigin,          // Indicates error location
    DWORD dwMessageId        // ERROR_XXX code value
)
{
    LPTSTR msgBuffer;        // string returned from system
    DWORD cBytes;           // length of returned string

    cBytes = FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_ALLOCATE_BUFFER,
        NULL,
        dwMessageId,
        MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
        (TCHAR *)&msgBuffer,
        500,
        NULL );

    if( msgBuffer )
    {
        msgBuffer[ cBytes ] = TEXT('\0');
    }
}

```

```
        printf( "Error: %s -- %s\n", lpOrigin, msgBuffer );
        LocalFree( msgBuffer );
    }
    else
    {
        printf( "FormatMessage error: %d\n", GetLastError());
    }
}
```