DYNAMIC ENGINEERING
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PHLnkBase
&
PHLnkChan

WDF Driver Documentation For the Six-Channel PCIe4L-HOTLink®

Developed with Windows Driver Foundation Ver1.9

Manual Revision B
Corresponding Firmware: Design ID 2, Revision C
Corresponding Hardware: 10-2013-0902

PHLnkBase, PHLnkChan
WDF Device Drivers for the
PCIe4L-HOTLink 6-Channel
HOTLink® Interface

Dynamic Engineering
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891
FAX: (831) 457-4793

Manual Revision B: Revised September 1, 2016

Table of Contents

Introduction

The PHLnkBase and PHLnkChan drivers are Windows device drivers for the PCIe4L-Six-Channel HOTLink design from Dynamic Engineering. These drivers were developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The HOTLink board has a Xilinx Spartan-6-LX100 FPGA to implement a PCI interface, FIFOs and protocol control/status for six HOTLink channels. There is a programmable PLL to create a custom Byte I/O clock from 16 to 32 MHz for the HOTLink I/O channels. The PCI bus is using a 50 MHz clock and interfaces with an onboard PCI-to-PCIe bridge that provides a four-lane PCIe interface.

Each channel has a 16k x 32-bit received data FIFO and an 8k x 32-bit transmit data FIFO implemented with FPGA internal RAM. These FIFOs can be accessed using either single-word reads or writes or DMA.

When the PCIe4L-HOTLink board is recognized by the PCI bus configuration utility it will load the PHLnkBase driver which will create a device object for each board, initialize the hardware, create child devices for the six I/O channels and request loading of the PHLnkChan driver. The PHLnkChan driver will create a device object for each of the I/O channels and perform initialization on each channel. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of DMA data in and out of the I/O channel devices.

Note

This documentation will provide information about all calls made to the drivers, and how the drivers interact with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PCIe4L-HOTLink hardware manual.

Driver Installation

There are several files provided in each driver package. These files include PHLnkBase.inf, PHLnkBase.cat, PHLnkBase.sys, PHLnkBasePublic.h, PHLnkChan.inf, PHLnkChan.cat, PHLnkChan.sys, PHLnkChanPublic.h and WdfCoInstaller01009.dll.

PHLnkBasePublic.h and PHLnkChanPublic.h are C header files that define the Application Program Interface (API) for the PHLnkBase and PHLnkChan drivers. These files are required at compile time by any application that wishes to interface with the drivers, but are not needed for driver installation.

Windows 7 Installation

Copy PHLnkBase.inf, PHLnkBase.cat, PHLnkBase.sys, PHLnkChan.inf, PHLnkChan.cat, PHLnkChan.sys and WdfCoInstaller01009.dll (Win7 version) to a removable memory device, or another accessible location if preferred.

With the PCIe4L-HOTLink hardware installed, power-on the PCIe host computer.

- Open the ***Device Manager*** from the control panel.
- Under ***Other devices*** there should be an ***Other PCI Bridge Device****.
- Right-click on the ***Other PCI Bridge Device*** and select ***Update Driver Software***.
- Insert the removable memory device prepared above if necessary.
- Select ***Browse my computer for driver software***.
- Select ***Let me pick from a list of device drivers on my computer***.
- Select ***Next***.
- Select ***Have Disk*** and enter the path where the driver files can be found.
- Select ***Next***.
- Select ***Close*** to close the update window.
  The system should now display the PHLnkChan I/O channels in the Device Manager.
- Right-click on each channel icon, select ***Update Driver Software*** and proceed as above for each channel as necessary.

***** If the ***Other PCI Bridge Device*** is not displayed, click on the ***Scan for hardware changes*** icon on the tool-bar.

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware. A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system. The interface to the device is identified using globally unique identifiers (GUID), which are defined in PHLnkBasePublic.h and PHLnkChanPublic.h. See main.c in the PcieHOTLinkUserApp project for an example of how to acquire handles for the base and six channel devices.

Note: In order to build an application you must link with setupapi.lib.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE        hDevice,          // Handle opened with CreateFile()
  DWORD         dwIoControlCode,  // Control code defined in API header file
  LPVOID        lpInBuffer,       // Pointer to input parameter
  DWORD         nInBufferSize,    // Size of input parameter
  LPVOID        lpOutBuffer,      // Pointer to output parameter
  DWORD         nOutBufferSize,   // Size of output parameter
  LPDWORD       lpBytesReturned,  // Pointer to return length parameter
  LPOVERLAPPED  lpOverlapped,     // Optional pointer to overlapped structure
);                                //   used for asynchronous I/O
```

The IOCTLs defined for the PHLnkBase driver are described below:

IOCTL_PHLNK_BASE_GET_INFO

Function: Returns the device driver version, design version, design type, user switch value, device instance number and PLL device ID.
Input: None
Output: PHLNK_BASE_DRIVER_DEVICE_INFO structure
Notes: The switch value is the configuration of the 8-bit onboard dipswitch that has been selected by the user (see the board silk screen for bit position and polarity). Instance number is the zero-based device number. See the definition of PHLNK_BASE_DRIVER_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _PHLNK_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignId;
   UCHAR    DesignRev;
   UCHAR    SwitchValue;
   UCHAR    PllDeviceId;
   UCHAR    NumChannels;   // 1..6
   UCHAR    InstanceNum;
} PHLNK_BASE_DRIVER_DEVICE_INFO, *PPHLNK_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_PHLNK_BASE_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input: PHLNK_BASE_PLL_DATA structure
Output: None
Notes: The PHLNK_BASE_PLL_DATA structure has only one field: Data ï an array of 40 bytes containing the PLL register data to write. See below for the definition of PHLNK_BASE_PLL_DATA.

```
#define PLL_MESSAGE1_SIZE  16
#define PLL_MESSAGE2_SIZE  24
#define PLL_MESSAGE_SIZE  (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _PHLNK_BASE_PLL_DATA {
   UCHAR    Data[PLL_MESSAGE_SIZE];
} PHLNK_BASE_PLL_DATA, *PPHLNK_BASE_PLL_DATA;
```

IOCTL_PHLNK_BASE_READ_PLL_DATA

Function: Returns the contents of the internal registers of the PLL.
Input: None
Output: PHLNK_BASE_PLL_DATA structure
Notes: The register data is written to the PHLNK_BASE_PLL_DATA structure in an array of 40 bytes. See definition of PHLNK_BASE_PLL_DATA above.

IOCTL_PHLNK_BASE_SET_COUNT_CONTROL

Function: Sets the group-start trigger counter control configuration.
Input: PHLNK_BASE_CHAN_START_CONFIG structure
Output: None
Notes: This call determines the group-start characteristics for enabled channel transmitters. If StartNow is true, then a trigger pulse is immediately sent to the channel transmitters. If CountEnable is true, the 20-bit counter starts counting at the rate of one megahertz. If ClearStart is true, the counter starts counting from zero, otherwise it loads the StartCount. When the counter reaches the TriggerCount, a trigger pulse is sent to the channel transmitters. If Continuous is true, the counter continues counting otherwise it stops when triggered. When the counter reaches the EndCount, the counter is re-initialized. If ClearEnable is true, the counter goes to zero, otherwise StartCount is loaded. See definition of PHLNK_BASE_CHAN_START_CONFIG below.

```
typedef struct _PHLNK_BASE_CHAN_START_CONFIG {
   BOOLEAN  StartNow;      // Start grouped transmit immediately
   BOOLEAN  CountEnable;   // Enable counter
   BOOLEAN  ClearStart;    // Clear count on start, else load StartCount
   BOOLEAN  ClearEnable;   // Clear count on rollover, else load StartCount
   BOOLEAN  Continuous;    // Counter runs continuously else single pass
   ULONG    StartCount;    // Preload count (0-0xffffff)
   ULONG    TriggerCount;  // Count to start transmissions (0-0xffffff)
   ULONG    EndCount;      // Rollover count (0-0xffffff)
} PHLNK_BASE_CHAN_START_CONFIG, *PPHLNK_BASE_CHAN_START_CONFIG;
```

IOCTL_PHLNK_BASE_GET_COUNT_CONTROL

Function: Returns the group-start trigger counter control configuration.
Input: None
Output: PHLNK_BASE_CHAN_START_CONFIG structure
Notes: Three 20-bit count registers and one control register are read and the information is returned in the PHLNK_BASE_PLL_DATA structure. See definition of PHLNK_BASE_CHAN_START_CONFIG above.

IOCTL_PHLNK_BASE_GET_ISR_STATUS

Function: Returns the accumulated status that was read in the ISR.
Input: None
Output: Interrupt status value (unsigned long integer)
Notes: This call was added to test some of the group-start capabilities of the design. When a frame done interrupt occurs this call is immediately made and the channels that have the group-start feature enabled should all show an interrupt status in the BASE_INT_CHAN_MASK field. Since these status bits are cleared in the channel DPC, the status bits are accumulated until the GetIsrStatus call is made so they will not be lost as the channel DPCs are run. When the GetIsrStatus is made, the status bits are cleared. See the Base status-bit field definitions below.

```
#define BASE_INT_CHAN_MASK 0x0000003F
#define BASE_PLL_WR_MASK   0x00000700
#define BASE_PLL_RD_MASK   0x00007000
#define BASE_PLL_STAT_MASK 0x00070000
#define BASE_CHAN_MASK     0x07000000
```

The IOCTLs defined for the PHLnkChan driver are described below:

IOCTL_PHLNK_CHAN_GET_INFO

Function: Returns the driver version and instance number of the device.
Input: None
Output: PHLNK_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of PHLNK_CHAN_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _PHLNK_CHAN_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    ChannelNum;
   UCHAR    SwitchValue;   // Board user switch value
   UCHAR    InstanceNum;   // Board instance from base driver
} PHLNK_CHAN_DRIVER_DEVICE_INFO, *PPHLNK_CHAN_DRIVER_DEVICE_INFO;
```

IOCTL_PHLNK_CHAN_SET_CONFIG

Function: Sets the requested channel control configuration.
Input: PHLNK_CHAN_CONFIG structure
Output: None
Notes: Specifies the enabled interrupt sources, DMA preemption behavior, transmit start mode, data storage mode and other control parameters. See the definitions of PHLNK_CHAN_CONFIG and its subordinate structures below.

```
typedef struct _PHLNK_CHAN_INTS {
   BOOLEAN  TxAmtInt;   // Transmit FIFO almost empty interrupt
   BOOLEAN  RxAflInt;   // Receive FIFO almost full interrupt
   BOOLEAN  RxOvflInt;  // Receive FIFO overflow interrupt
   BOOLEAN  TxFrmDnInt; // Transmit frame done interrupt
   BOOLEAN  RxFrmDnInt; // Receive frame done interrupt
} PHLNK_CHAN_INTS, *PPHLNK_CHAN_INTS;

 // Channel DMA priority (use sparingly)
typedef enum _PHLNK_DMA_PRMPT {
   PHLNK_NONE,     // No priority
   PHLNK_READ,     // Read DMA has priority
   PHLNK_WRITE,    // Write DMA has priority
   PHLNK_RDWR      // Read and Write DMA have priority
} PHLNK_DMA_PRMPT, *PPHLNK_DMA_PRMPT;

 // Channel Receiver storage mode
typedef enum _PHLNK_RX_MODE {
   STORE_ALL,  // Store data and all control
   DATA_ONLY,  // Store data only
   SINGLE_CTRL,// Store data and non-repeated control
} PHLNK_RX_MODE, *PPHLNK_RX_MODE;

 // Channel Transmitter start mode
typedef enum _PHLNK_TX_START {
   VIDEO_FRM,  // Video frame mode
   SYNC_NONE,  // Ignore group-sync signal
   SYNC_FIRST, // Synchronize first frame with sync signal
   SYNC_ALL,   // Synchronize all frames with sync signal
} PHLNK_TX_START, *PPHLNK_TX_START;
```

```
typedef struct _PHLNK_CHAN_CONFIG {
   BOOLEAN            TxEnable;    // Enable HOTLink transmitter
   BOOLEAN            RxEnable;    // Enable HOTLink receiver
   BOOLEAN            FifoTestEn;  // Enables auto tx->rx FIFO transfer
   BOOLEAN            TxOutEn;     // Enable transmitter output
   BOOLEAN            TxBitEn;     // Built-in-test enable (sends test pattern)
   BOOLEAN            TxLdEn;      // Enables loading of built-in-test data
   BOOLEAN            TxClearEn;   // Enables clearing Tx Frame request when frame done
   BOOLEAN            RxInASel;    // Selects Rx input '1'=External, '0'=Local Tx
   BOOLEAN            RxBitEn;     // Built-in-test enable (verifies test pattern)
   BOOLEAN            RxReframe;   // '1'=K28.5 re-syncs data, '0'=sync locked
   BOOLEAN            RxTestEn;    // Forces the receiver to start immediately
   BOOLEAN            MuxEnable;   // '1'=Enable Tx/Rx Mux, '0'=Mux disabled
   BOOLEAN            TxSelect;    // '1'=Transmit selected, '0'=Receive selected
   BOOLEAN            NoStopSeq;   // '1'=No stop sequence, '0'=Send stop sequence
   PHLNK_TX_START     TxStart;     // Video frame or various stream synch options
   PHLNK_RX_MODE      RxStoreMd;   // Receiver storage mode
   PHLNK_CHAN_INTS    IntConfig;   // Interrupt condition enables
   PHLNK_DMA_PRMPT    DmaPriority;// DMA preemption control
} PHLNK_CHAN_CONFIG, *PPHLNK_CHAN_CONFIG;
```

IOCTL_PHLNK_CHAN_GET_CONFIG

Function: Returns the fields set in the previous call.
Input: None
Output: PHLNK_CHAN_CONFIG structure
Notes: See the definitions of PHLNK_INTS, PHLNK_DMA_PRMPT, PHLNK_RX_MODE, PHLNK_TX_START and PHLNK_CHAN_CONFIG above.

IOCTL_PHLNK_CHAN_GET_STATUS

Function: Returns the channelôs status register value and clears the latched status bits.
Input: None
Output: Value of the channelôs status register (unsigned long integer)
Notes: See the status bit definitions below. Only the bits in CHAN_STAT_MASK will be returned. The bits in CHAN_STAT_LATCH_MASK will be cleared by this call only if they are set when the register was read. This prevents the possibility of missing an interrupt condition that occurs after the register has been read but before the latched register bits are cleared.

```
// Status bit definitions
#define CHAN_STAT_TX_FF_MT       0x00000001
#define CHAN_STAT_TX_FF_AMT      0x00000002
#define CHAN_STAT_TX_FF_FL       0x00000004
#define CHAN_STAT_TX_FF_VLD      0x00000008
#define CHAN_STAT_RX_FF_MT       0x00000010
#define CHAN_STAT_RX_FF_AFL      0x00000020
#define CHAN_STAT_RX_FF_FL       0x00000040
#define CHAN_STAT_RX_FF_VLD      0x00000080
#define CHAN_STAT_TX_AMT_INT     0x00000100
#define CHAN_STAT_RX_AFL_INT     0x00000200
#define CHAN_STAT_RX_OVFL        0x00000400
#define CHAN_STAT_RX_SYM_ERR     0x00000800
#define CHAN_STAT_WR_DMA_INT     0x00001000
```

```
#define CHAN_STAT_RD_DMA_INT      0x00002000
#define CHAN_STAT_WR_DMA_ERR      0x00004000
#define CHAN_STAT_RD_DMA_ERR      0x00008000
#define CHAN_STAT_WR_DMA_RDY      0x00010000
#define CHAN_STAT_RD_DMA_RDY      0x00020000
#define CHAN_STAT_RX_DATA_RDY     0x00040000
#define CHAN_STAT_TX_DATA_READ    0x00080000
#define CHAN_STAT_TX_FRAME_DN     0x00100000
#define CHAN_STAT_RX_FRAME_DN     0x00200000
#define CHAN_STAT_RX_ACTIVE       0x00400000
#define CHAN_STAT_RXIO_FF_FL      0x00800000
#define CHAN_STAT_AUX_FF_MT       0x01000000
#define CHAN_STAT_AUX_FF_AFL      0x02000000
#define CHAN_STAT_AUX_FF_FL       0x04000000
#define CHAN_STAT_TX_DAT_VLD      0x08000000
#define CHAN_STAT_TX_VLD_MASK     0x30000000
#define CHAN_STAT_LOC_INT         0x40000000
#define CHAN_STAT_INT_ACTIVE      0x80000000
```

IOCTL_PHLNK_CHAN_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: PHLNK_CHAN_FIFO_LEVELS structure
Output: None
Notes: These values are initialized to the default values à FIFO and Õ FIFO respectively when the driver initializes. The FIFO counts are compared to these levels to set the state of the CHAN_STAT_TX_FF_AMT and CHAN_STAT_RX_FF_AFL status bits and latch the CHAN_STAT_TX_AMT_LT and CHAN_STAT_RX_AFL_LT latched status bits. Also if the control bits CHAN_CNTRL_URGNT_OUT_EN and/or CHAN_CNTRL_URGNT_IN_EN are set, the FIFO level values are used to determine when to give priority to an output or input DMA channel that is running out of data or room to store data. See the definition of PHLNK_CHAN_FIFO_LEVELS below.

```
typedef struct _PHLNK_CHAN_FIFO_LEVELS {
   ULONG    AlmostFull;
   ULONG    AlmostEmpty;
} PHLNK_CHAN_FIFO_LEVELS, *PPHLNK_CHAN_FIFO_LEVELS;
```

IOCTL_PHLNK_CHAN_GET_FIFO_LEVELS

Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: PHLNK_CHAN_FIFO_LEVELS structure
Notes: Returns the values set in the previous call.

IOCTL_PHLNK_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive data FIFOs.
Input: None
Output: PHLNK_CHAN_FIFO_COUNTS structure
Notes: There are three pipe-line latches and a fifteen word auxiliary FIFO in addition to the 8k data FIFO for the transmit data-path and four pipe-line latches and a 16k data FIFO for the receive data-path. These are counted in the FIFO counts. That means the transmit count can be a maximum of 8210 32-bit words and the receive count can be a maximum of 16388 32-bit words. See the definition of PHLNK_CHAN_FIFO_COUNTS below.

```
typedef struct _PHLNK_CHAN_FIFO_COUNTS {
   ULONG   TxCount;
   ULONG   RxCount;
} PHLNK_CHAN_FIFO_COUNTS, *PPHLNK_CHAN_FIFO_COUNTS;
```

IOCTL_PHLNK_CHAN_RESET_FIFOS

Function: Resets one or both FIFOs for the referenced channel.
Input: PHLNK_FIFO_SEL enumeration type
Output: None
Notes: Resets the transmit or receive FIFO or both depending on the input parameter selection. See the definition of PHLNK_CHAN_FIFO_SEL below.

```
 // Used for FIFO reset call
typedef enum _PHLNK_CHAN_FIFO_SEL {
   PHLNK_TX,
   PHLNK_RX,
   PHLNK_BOTH
} PHLNK_CHAN_FIFO_SEL, *PPHLNK_CHAN_FIFO_SEL;
```

IOCTL_PHLNK_CHAN_WRITE_FIFO

Function: Writes a 32-bit data-word to the transmit FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_PHLNK_CHAN_READ_FIFO

Function: Returns a 32-bit data word from the receive FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to make single-word accesses to the receive FIFO instead of using DMA.

IOCTL_PHLNK_CHAN_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt. The DMA interrupts do not cause this event to be signaled.

IOCTL_PHLNK_CHAN_ENABLE_INTERRUPT

Function: Enables the channel master interrupt.
Input: None
Output: None
Notes: This command must be run to allow the board to respond to user interrupts. The master interrupt enable is disabled in the driver interrupt service routine when a user interrupt is serviced. Therefore this command must be run after each user interrupt occurs to re-enable it.

IOCTL_PHLNK_CHAN_DISABLE_INTERRUPT

Function: Disables the channel master interrupt.
Input: None
Output: None
Notes: This call is used when user interrupt processing is no longer desired.

IOCTL_PHLNK_CHAN_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_PHLNK_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (unsigned long integer)
Notes: Returns the status that was read while servicing the last interrupt caused by one of the user-enabled channel interrupt conditions. The interrupts that deal with the DMA transfers do not affect this value. The new field is true if the stored ISR status has been updated since the last time this call was made. See below for the definition of PHLNK_CHAN_ISR_STATUS.

```
typedef struct _PHLNK_CHAN_ISR_STATUS {
   ULONG    Status;  // Value of status register read in ISR
   BOOLEAN  New;     // True if the status has changed since last GetIsrStatus call
} PHLNK_CHAN_ISR_STATUS, *PPHLNK_CHAN_ISR_STATUS;
```

IOCTL_PHLNK_CHAN_SET_IO_PARAMS

Function: Sets the start and stop sequences and byte count for I/O transfers.
Input: PHLNK_CHAN_IO_PARAMS structure
Output: None
Notes: Start and Stop sequences are inserted by the transmitter to mark the beginning and end of a data-frame. The receiver uses these sequences to determine when to start storing data, when to stop, and for detecting byte-counts for each data-frame. A new field, FrmSpcr was added to control subsequent frame timing in the sync initial frame transmit start mode. This count determines the number of idle bytes sent between frames. See the definitions of PHLNK_CHAN_IO_CHAR and PHLNK_CHAN_IO_PARAMS below.

```
typedef struct _PHLNK_CHAN_IO_CHAR {
   BOOLEAN CntrlChar;
   UCHAR   Byte;
} PHLNK_CHAN_IO_CHAR, *PPHLNK_CHAN_IO_CHAR;

#define MAX_CHARS_PER_SEQ  3

 // Defaults loaded if fields are zero
 // (Start-0x105,0x104; Stop-0x104,0x105; Count-0x008000)
typedef struct _PHLNK_CHAN_IO_PARAMS {
   UCHAR                StartCnt;              // Zero to three characters
   UCHAR                StopCnt;               // Zero to three characters
   PHLNK_CHAN_IO_CHAR   StartSeq[MAX_CHARS_PER_SEQ];
   PHLNK_CHAN_IO_CHAR   StopSeq[MAX_CHARS_PER_SEQ];
   ULONG                ByteCnt;               // 16 MByte max count (24 bits)
   ULONG                FrmSpcr;               // 16 MByte max count (24 bits)
} PHLNK_CHAN_IO_PARAMS, *PPHLNK_CHAN_IO_PARAMS;
```

IOCTL_PHLNK_CHAN_GET_IO_PARAMS

Function: Returns the start and stop sequences, byte count and inter-frame spacer for I/O transfers.
Input: None
Output: PHLNK_CHAN_IO_PARAMS structure
Notes: Returns the values set in the previous call. See structure definitions above.

Write

HOTLink DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the I/O channel device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Read

HOTLink DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the I/O channel device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Warranty and Repair

http://www.dyneng.com/warranty.html

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be ñcockpit errorò rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customerôs making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customerôs invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite C Santa Cruz, CA 95060
(831) 457-8891 Fax (831) 457-4793
support@dyneng.com