**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

# PHLRv1Base
# &
# PHLRv1Chan

## WDF Driver Documentation For the Five-Channel PCIe4L-HOTLink®

**Developed with Windows Driver Foundation Ver1.9**

Manual Revision A
Corresponding Firmware: Design ID 3, Revision A
Corresponding Hardware: 10-2016-2801

**PHLRv1Base, PHLRv1Chan**
WDF Device Drivers for the PCIe4L-HOTLink 5-Channel HOTLink® Interface

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

Manual Revision A: Revised May 17, 2017

# Table of Contents

**Introduction** .... 4
**Note** .... 4
**Driver Installation** .... 4
**Windows 7 Installation** .... 5
**Driver Startup** .... 5
**IO Controls** .... 6
IOCTL_PHLRV1_BASE_GET_INFO .... 6
IOCTL_PHLRV1_CHAN_GET_INFO .... 8
IOCTL_PHLRV1_CHAN_SET_CONFIG .... 8
IOCTL_PHLRV1_CHAN_GET_CONFIG .... 9
IOCTL_PHLRV1_CHAN_GET_TX_FRAME_COUNTS .... 9
IOCTL_PHLRV1_CHAN_GET_RX_FRAME_COUNTS .... 9
IOCTL_PHLRV1_CHAN_GET_STATUS .... 10
IOCTL_PHLRV1_CHAN_SET_FIFO_LEVELS .... 11
IOCTL_PHLRV1_CHAN_GET_FIFO_LEVELS .... 11
IOCTL_PHLRV1_CHAN_GET_FIFO_COUNTS .... 11
IOCTL_PHLRV1_CHAN_RESET_FIFOS .... 12
IOCTL_PHLRV1_CHAN_WRITE_FIFO .... 12
IOCTL_PHLRV1_CHAN_READ_FIFO .... 12
IOCTL_PHLRV1_CHAN_REGISTER_EVENT .... 12
IOCTL_PHLRV1_CHAN_ENABLE_INTERRUPT .... 13
IOCTL_PHLRV1_CHAN_DISABLE_INTERRUPT .... 13
IOCTL_PHLRV1_CHAN_FORCE_INTERRUPT .... 13
IOCTL_PHLRV1_CHAN_GET_ISR_STATUS .... 13
IOCTL_PHLRV1_CHAN_CLEAR_RVS_COUNT .... 14
IOCTL_PHLRV1_CHAN_GET_LINK_STATUS .... 14
IOCTL_PHLRV1_CHAN_WRITE_SFP_FIFO .... 14
IOCTL_PHLRV1_CHAN_READ_SFP_FIFO .... 14
IOCTL_PHLRV1_CHAN_SFP_ACCESS .... 15
IOCTL_PHLRV1_CHAN_TEST_SFP .... 15
**Write** .... 16
**Read** .... 16
**Warranty and Repair** .... 17
**Service Policy** .... 17
Out of Warranty Repairs .... 17
**For Service Contact:** .... 17

## Introduction

The PHLRv1Base and PHLRv1Chan 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 five HOTLink 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 FIFO for received data and an 8k x 32-bit FIFO for transmit data implemented with FPGA internal RAM. These FIFOs can be accessed using either single-word writes/reads or DMA.

When the PCIe4L-HOTLink-X5 board is recognized by the PCI bus configuration utility it will load the PHLRv1Base driver which will create a device object for each board, initialize the hardware, create child devices for the five I/O channels and request loading of the PHLRv1Chan driver. The PHLRv1Chan 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-X5 hardware manual.

## Driver Installation

There are several files provided in each driver package. These files include PHLRv1Base.inf, PHLRv1Base.cat, PHLRv1Base.sys, PHLRv1BasePublic.h, PHLRv1Chan.inf, PHLRv1Chan.cat, PHLRv1Chan.sys, PHLRv1ChanPublic.h and WdfCoInstaller01009.dll.

PHLRv1BasePublic.h and PHLRv1ChanPublic.h are C header files that define the Application Program Interface (API) for the PHLRv1Base and PHLRv1Chan 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 PHLRv1Base.inf, PHLRv1Base.cat, PHLRv1Base.sys, PHLRv1Chan.inf, PHLRv1Chan.cat, PHLRv1Chan.sys and WdfCoInstaller01009.dll (Win7 version) to a removable memory device, or if preferred, another system accessible location.

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 ***Browse*** and navigate to the location where the driver files can be found
- Select ***Next***.
- If a Windows Security alert dialog bow is displayed, select ***Install***.
- Select ***Close*** to close the update window.
  The system should now display the PHLRv1Chan 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 drivers have been installed they 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 PHLRv1BasePublic.h and PHLRv1ChanPublic.h. See main.c in the PcieHLRv1UserApp project for an example of how to acquire handles for the base and five 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 PHLRv1Base driver are described below:**

### IOCTL_PHLRV1_BASE_GET_INFO

***Function:*** Returns the device driver revision, design revision, design ID, user switch value and device instance number.
***Input:*** None
***Output:*** PHLRV1_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 PHLRV1_BASE_DRIVER_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _PHLRV1_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignId;
   UCHAR    DesignRev;
   UCHAR    SwitchValue;
   UCHAR    NumChannels;   // 1..5
   UCHAR    InstanceNum;
} PHLRV1_BASE_DRIVER_DEVICE_INFO, *PPHLRV1_BASE_DRIVER_DEVICE_INFO;
```

## IOCTL_PHLRV1_BASE_GET_STATUS

***Function:*** Returns the base status register value.
***Input:*** None
***Output:*** Value of the base status register (unsigned long integer)
***Notes:*** A ‘1’ in any of the low five bits signifies that an interrupt is active for the respective channel. The channel number mask bits report the number of the last channel implemented. See the status bit definitions below.

```
 // Status bit definitions
#define BASE_INT_CHAN_0       0x00000001
#define BASE_INT_CHAN_1       0x00000002
#define BASE_INT_CHAN_2       0x00000004
#define BASE_INT_CHAN_3       0x00000008
#define BASE_INT_CHAN_4       0x00000010

#define BASE_INT_CHAN_MASK    0x0000001F

#define BASE_CHAN_NUM_MASK    0x07000000  // Last channel installed

#define BASE_STATUS_MASK     (BASE_INT_CHAN_MASK | BASE_CHAN_NUM_MASK)
```

**The IOCTLs defined for the PHLRv1Chan driver are described below:**

## IOCTL_PHLRV1_CHAN_GET_INFO

***Function:*** Returns the driver revision, channel number, user switch setting and instance number of the device.
***Input:*** None
***Output:*** PHLRV1_CHAN_DRIVER_DEVICE_INFO structure
***Notes:*** Instance number and switch value are passed to the channel driver from the base driver. Instance number is the board instance number. This is used to determine which board the channel device belongs to if there is more than one PCIe4L-HOTLink-RV1 board installed. See the definition of PHLRV1_CHAN_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _PHLRV1_CHAN_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    ChannelNum;
   UCHAR    SwitchValue;   // Board user switch value
   UCHAR    InstanceNum;   // Board instance from base driver
} PHLRV1_CHAN_DRIVER_DEVICE_INFO, *PPHLRV1_CHAN_DRIVER_DEVICE_INFO;
```

## IOCTL_PHLRV1_CHAN_SET_CONFIG

***Function:*** Sets the target channel control configuration.
***Input:*** PHLRV1_CHAN_CONFIG structure
***Output:*** None
***Notes:*** Specifies the enabled interrupt sources, DMA preemption behavior, I/O data path, enables and other control parameters. See the definitions of PHLRV1_CHAN_CONFIG and its subordinate structures below.

```
typedef struct _PHLRV1_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           TxGo;       // Start transmission
   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           TxBigEndEn; // Transmit data Big Endian enable
   BOOLEAN           RxBigEndEn; // Received data Big Endian enable
   PHLRV1_CHAN_INTS  IntConfig;  // Interrupt condition enables
   PHLRV1_DMA_PRMPT  DmaPriority;// DMA preemption control
} PHLRV1_CHAN_CONFIG, *PPHLRV1_CHAN_CONFIG;
```

```
typedef struct _PHLRV1_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
} PHLRV1_CHAN_INTS, *PPHLRV1_CHAN_INTS;

 // Channel DMA priority (use sparingly)
typedef enum _PHLRV1_DMA_PRMPT {
   PHLRV1_NONE,     // No priority
   PHLRV1_READ,     // Read DMA has priority
   PHLRV1_WRITE,    // Write DMA has priority
   PHLRV1_RDWR      // Read and Write DMA have priority
} PHLRV1_DMA_PRMPT, *PPHLRV1_DMA_PRMPT;
```

### IOCTL_PHLRV1_CHAN_GET_CONFIG

***Function:*** Returns the fields set in the previous call.
***Input:*** None
***Output:*** PHLRV1_CHAN_CONFIG structure
***Notes:*** See the definitions of PHLRV1_INTS, PHLRV1_DMA_PRMPT and PHLRV1_CHAN_CONFIG above.

### IOCTL_PHLRV1_CHAN_GET_TX_FRAME_COUNTS

***Function:*** Returns the number of bytes in the header and data sections of the most recently transmitted data-frame.
***Input:*** None
***Output:*** PHLRV1_CHAN_FRAME_COUNTS structure
***Notes:*** See the definition of PHLRV1_CHAN_FRAME_COUNTS below.

```
typedef struct _PHLRV1_CHAN_FRAME_COUNTS {
   ULONG    HeaderByteCount;
   ULONG    DataByteCount;
} PHLRV1_CHAN_FRAME_COUNTS, *PPHLRV1_CHAN_FRAME_COUNTS;
```

### IOCTL_PHLRV1_CHAN_GET_RX_FRAME_COUNTS

***Function:*** Returns the number of bytes in the header and data sections of the most recently received data-frame.
***Input:*** None
***Output:*** PHLRV1_CHAN_FRAME_COUNTS structure
***Notes:*** See the definition of PHLRV1_CHAN_FRAME_COUNTS above.

## IOCTL_PHLRV1_CHAN_GET_STATUS

***Function:*** Returns the channel's status register value and clears the latched status bits.
***Input:*** None
***Output:*** Value of the channel's 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_RX_SYNCHED          0x00800000
#define CHAN_STAT_TX_FRM_ERR          0x01000000
#define CHAN_STAT_RX_FRM_ERR          0x02000000
#define CHAN_STAT_SFP_PRG_ERR         0x04000000
#define CHAN_STAT_SFP_PRG_DN          0x08000000
#define CHAN_STAT_SFP_RDY             0x10000000
#define CHAN_STAT_RX_IO_FULL          0x20000000
#define CHAN_STAT_LOC_INT             0x40000000
#define CHAN_STAT_INT_ACTIVE          0x80000000
```

### IOCTL_PHLRV1_CHAN_SET_FIFO_LEVELS

***Function:*** Sets the transmitter almost empty and receiver almost full levels for the channel.
***Input:*** PHLRV1_CHAN_FIFO_LEVELS structure
***Output:*** None
***Notes:*** These values are initialized to the default values ⅛ FIFO and ⅞ 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. If DMA preemption is enabled in the channel configuration, these 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. When one of these conditions becomes true the CHAN_STAT_TX_AMT_INT or CHAN_STAT_RX_AFL_INT status bits will be latched. These bits are used for FIFO level interrupt processing. See the definition of PHLRV1_CHAN_FIFO_LEVELS below.

```
typedef struct _PHLRV1_CHAN_FIFO_LEVELS {
   ULONG    AlmostFull;
   ULONG    AlmostEmpty;
} PHLRV1_CHAN_FIFO_LEVELS, *PPHLRV1_CHAN_FIFO_LEVELS;
```

### IOCTL_PHLRV1_CHAN_GET_FIFO_LEVELS

***Function:*** Returns the transmitter almost empty and receiver almost full levels for the channel.
***Input:*** None
***Output:*** PHLRV1_CHAN_FIFO_LEVELS structure
***Notes:*** Returns the values set in the previous call.

### IOCTL_PHLRV1_CHAN_GET_FIFO_COUNTS

***Function:*** Returns the number of data words in the transmitter and receiver data FIFOs.
***Input:*** None
***Output:*** PHLRV1_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-1 data FIFO for the receive data-path. These are all counted in the transmit and receive FIFO counts. Hence the transmit count can be a maximum of 8210 32-bit words and the receive count can be a maximum of 16387 32-bit words. See the definition of PHLRV1_CHAN_FIFO_COUNTS below.

```
typedef struct _PHLRV1_CHAN_FIFO_COUNTS {
   ULONG   TxCount;
   ULONG   RxCount;
} PHLRV1_CHAN_FIFO_COUNTS, *PPHLRV1_CHAN_FIFO_COUNTS;
```

## IOCTL_PHLRV1_CHAN_RESET_FIFOS

***Function:*** Resets one or both FIFOs for the referenced channel.
***Input:*** PHLRV1_FIFO_SEL enumeration type
***Output:*** None
***Notes:*** Resets the transmitter or receiver FIFO or both depending on the input parameter selection.  See the definition of PHLRV1_CHAN_FIFO_SEL below.

```
 // Used for FIFO reset call
typedef enum _PHLRV1_CHAN_FIFO_SEL {
   PHLRV1_TX,
   PHLRV1_RX,
   PHLRV1_BOTH
} PHLRV1_CHAN_FIFO_SEL, *PPHLRV1_CHAN_FIFO_SEL;
```

## IOCTL_PHLRV1_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_PHLRV1_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_PHLRV1_CHAN_REGISTER_EVENT

***Function:*** Registers an event to be signaled when a user 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's 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_PHLRV1_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_PHLRV1_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_PHLRV1_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_PHLRV1_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 as a result of a serviced interrupt since the last time this call was made. See below for the definition of PHLRV1_CHAN_ISR_STATUS. The status bit definitions were listed in the description of the IOCTL_PHLRV1_CHAN_GET_STATUS call.

```
typedef struct _PHLRV1_CHAN_ISR_STATUS {
   ULONG    Status;  // Value of status register read in ISR
   BOOLEAN  New;     // True if the status has changed since last GetIsrStatus call
} PHLRV1_CHAN_ISR_STATUS, *PPHLRV1_CHAN_ISR_STATUS;
```

## IOCTL_PHLRV1_CHAN_CLEAR_RVS_COUNT

***Function:*** Clears the RVS counter.
***Input:*** None
***Output:*** None
***Notes:*** The Received Violation Symbol counter is incremented whenever a symbol error is detected by the receiver. The RVS count field is 24 bits wide. The count will be reset to zero when this call is made.

## IOCTL_PHLRV1_CHAN_GET_LINK_STATUS

***Function:*** Returns the values read from the SFP link status register.
***Input:*** None
***Output:*** PHLRV1_CHAN_LINK_STATUS structure
***Notes:*** Returns the values read from the SFP link status register. See structure definition below.

```
typedef struct _PHLRV1_CHAN_LINK_STATUS {
   BOOLEAN  NoSfp;       // SFP module not installed
   BOOLEAN  TxFault;     // Transmitter fault
   BOOLEAN  RxSigLost;   // Receiver loss-of-signal
   BOOLEAN  SfpDatin;    // Reports the level of the sdat input
   BOOLEAN  SfpWrtFull;  // SFP programming data FIFO is full
   BOOLEAN  SfpWrtEmpty;// SFP programming data FIFO is empty
   BOOLEAN  SfpRdFull;   // SFP read data FIFO is full
   BOOLEAN  SfpRdEmpty;  // SFP read data FIFO is empty
   ULONG    RvsCount;    // Count of received violation symbols
} PHLRV1_CHAN_LINK_STATUS, *PPHLRV1_CHAN_LINK_STATUS;
```

## IOCTL_PHLRV1_CHAN_WRITE_SFP_FIFO

***Function:*** Writes a 32-bit data-word to the SFP write FIFO.
***Input:*** FIFO word (unsigned long integer)
***Output:*** None
***Notes:*** Data written to this FIFO is read by the SFP programmer when it is commanded to write to the internal memory of the SFP for the referenced channel. Starting at the stored address offset, the programmer writes the specified number of bytes sequentially to the SFP.

## IOCTL_PHLRV1_CHAN_READ_SFP_FIFO

***Function:*** Returns a 32-bit data word from the SFP read FIFO.
***Input:*** None
***Output:*** FIFO word (unsigned long integer)
***Notes:*** When a read request is made to the SFP programmer, starting at the stored address offset, the specified number of bytes are read and assembled into long words and written to the SFP read FIFO. This call reads and returns data words from that FIFO.

## IOCTL_PHLRV1_CHAN_SFP_ACCESS

***Function:*** Tasks the SFP programmer to write or read data to/from the SFP module.
***Input:*** PHLRV1_CHAN_SFP_CNTRL structure
***Output:*** None
***Notes:*** Address is a byte address. When SfpDiagEn is true, the device ID sent is 0xA2, otherwise the device ID is 0xA0. These ID values access different memory spaces. See the definition of PHLRV1_CHAN_SFP_CNTRL below.

```
typedef struct _PHLRV1_CHAN_SFP_CNTRL {
   UCHAR    Address;    // Address offset
   UCHAR    ByteCount;  // Number of bytes to read/write
   BOOLEAN  SfpDiagEn;  // When true use diagnostic ID (0xA2)
   BOOLEAN  SfpReadEn;  // Read when true, Write when false
} PHLRV1_CHAN_SFP_CNTRL, *PPHLRV1_CHAN_SFP_CNTRL;
```

## IOCTL_PHLRV1_CHAN_TEST_SFP

***Function:*** Enables or disables the test function for the SFP control and status signals.
***Input:*** Enable (BOOLEAN)
***Output:*** None
***Notes:*** When enabled, a 10 MHz clock is driven onto the SClk output pin. The RV1 test fixture allows this output pin to be connected to one of four input pins: Rx Signal Lost, Tx Fault, Module Not Installed or SDataIn. When the clock is detected on one of these status inputs, the continuity of that signal is confirmed. The SFP module is external to the board and not available for acceptance testing. This test function allows the integrity of the control and status signals to be verified.

## 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 “cockpit error” 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 customer’s 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 customer’s 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