PmcHLnkBase & PmcHLnkChan

WDF Driver Documentation For the Two-Channel ccPMC-HOTLink-Kaon1

Developed with Windows Driver Foundation Ver1.9

Manual Revision B
Corresponding Firmware: Design ID 4, Revision B2
Corresponding Hardware: 10-2009-0103
**PmcHLnkBase, PmcHLnkChan**
WDF Device Drivers for the ccPMC-HOTLink-Kaon1 2-Channel HOTLink® Interface

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

Dynamic Engineering’s products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Connection of incompatible hardware is likely to cause serious damage.
Table of Contents

Introduction ........................................................................................................................................... 4
Note ...................................................................................................................................................... 4
Driver Installation ................................................................................................................................. 4
Windows 7 Installation .......................................................................................................................... 5
Driver Startup ......................................................................................................................................... 5
IO Controls ............................................................................................................................................. 6
  IOCTL_PMC_HLNK_BASE_GET_INFO ............................................................................................... 6
  IOCTL_PMC_HLNK_BASE_LOAD_PLL_DATA ....................................................................................... 7
  IOCTL_PMC_HLNK_BASE_READ_PLL_DATA ....................................................................................... 7
  IOCTL_PMC_HLNK_BASE_GET_STATUS .............................................................................................. 7
  IOCTL_PMC_HLNK_CHAN_GET_INFO ................................................................................................. 8
  IOCTL_PMC_HLNK_CHAN_SET_CONFIG ............................................................................................. 8
  IOCTL_PMC_HLNK_CHAN_GET_CONFIG ............................................................................................. 9
  IOCTL_PMC_HLNK_CHAN_GET_STATUS ............................................................................................ 9
  IOCTL_PMC_HLNK_CHAN_SET_FIFO_LEVELS ................................................................................... 10
  IOCTL_PMC_HLNK_CHAN_GET_FIFO_LEVELS ................................................................................... 10
  IOCTL_PMC_HLNK_CHAN_GET_FIFO_COUNTS .................................................................................. 10
  IOCTL_PMC_HLNK_CHAN_RESET_FIFOS ......................................................................................... 11
  IOCTL_PMC_HLNK_CHAN_WRITE_FIFO ............................................................................................. 11
  IOCTL_PMC_HLNK_CHAN_READ_FIFO ............................................................................................... 11
  IOCTL_PMC_HLNK_CHAN_WRITE_RAM ............................................................................................... 11
  IOCTL_PMC_HLNK_CHAN_READ_RAM ............................................................................................... 11
  IOCTL_PMC_HLNK_CHAN_GET_MSG_COUNTS ................................................................................... 12
  IOCTL_PMC_HLNK_CHAN_SET_TTL_CONFIG ...................................................................................... 12
  IOCTL_PMC_HLNK_CHAN_GET_TTL_CONFIG ...................................................................................... 12
  IOCTL_PMC_HLNK_CHAN_SET_TTL_FIFO ......................................................................................... 13
  IOCTL_PMC_HLNK_CHAN_GET_TTL_FIFO ......................................................................................... 13
  IOCTL_PMC_HLNK_CHAN_RESET_TTL_FIFOS .................................................................................... 14
  IOCTL_PMC_HLNK_CHAN_WRITE_TTL_FIFO ...................................................................................... 14
  IOCTL_PMC_HLNK_CHAN_READ_TTL_FIFO ....................................................................................... 14
  IOCTL_PMC_HLNK_CHAN_REGISTER_EVENT .................................................................................... 14
  IOCTL_PMC_HLNK_CHAN_ENABLE_INTERRUPT ................................................................................... 15
  IOCTL_PMC_HLNK_CHAN_DISABLE_INTERRUPT ............................................................................... 15
  IOCTL_PMC_HLNK_CHAN_FORCE_INTERRUPT .................................................................................... 15
  IOCTL_PMC_HLNK_CHAN_GET_ISR_STATUS ....................................................................................... 15
  IOCTL_PMC_HLNK_CHAN_READ_DMA_COUNTS ................................................................................... 16
Write ......................................................................................................................................................... 16
Read ......................................................................................................................................................... 16
Warranty and Repair .............................................................................................................................. 17
Service Policy ......................................................................................................................................... 17
Out of Warranty Repairs ....................................................................................................................... 17
For Service Contact ............................................................................................................................... 17
Introduction

The PmcHLnkBase and PmcHLnkChan drivers are Windows device drivers for the ccPMC-Two-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 two HOTLink channels. There is a programmable PLL to create a custom Byte I/O clock of 16.777216 MHz for the HOTLink GDL (Global Data Link) interface, a 4x clock of 27.525120 MHz for the GCS (Global Clock Sync) transmitter and a 147.456 MHz sample clock for the GCS receiver. The PCI bus is using a 33 MHz clock and interfaces with the host PCI bus through a PCI to PMC carrier board.

Each channel's GDL has two 32k x 32-bit FIFOs one each for the transmitter and receiver. These FIFOs can be accessed using either single-word reads / writes or DMA. Each channel’s GDL also has a 128 x 32-bit RAM block to store format information for the GDL data-frame. The format RAM is loaded with single word writes and accessed by both the GDL transmitter and receiver during the transmission and reception of GDL data.

The GCS has two 4k x 32-bit FIFOs one each for the transmitter and receiver. These FIFOs are accessible only by single-word writes and reads. All FIFOs and RAM are implemented using FPGA internal RAM blocks.

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

Driver Installation

There are several files provided in each driver package. These files include PmcHLnkBase.cat, PmcHLnkBase.sys, PmcHLnkBase.inf, PmcHLnkChan.cat, PmcHLnkChan.sys, PmcHLnkChan.inf, and WdfCoInstaller01009.dll.
PmcHLnkBasePublic.h and PmcHLnkChanPublic.h are C header files that define the Application Program Interface (API) for the PmcHLnkBase and PmcHLnkChan drivers. These two 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 PmcHLnkBase.inf, PmcHLnkBase.cat, PmcHLnkBase.sys, PmcHLnkChan.inf, PmcHLnkChan.cat, PmcHLnkChan.sys and WdfCoInstaller01009.dll (Win7 version) to a removable memory device, or another accessible location if preferred.

With the ccPMC-HOTLink hardware installed, power-on the PCI 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**.
- Navigate to the location of the prepared memory device or wherever the specified files are located.
- Select **Next**.
- If a dialog box is displayed asking if you are sure that you want to install the driver Select **Yes**.
- Select **Next**.
- Select **Close** to close the update window. The system should now display the PmcHLnkChan 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 PmcHLnkBasePublic.h and PmcHLnkChanPublic.h. See main.c in the example PmcHOTLinkUserApp project for information about how to acquire handles for the base and two 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.

```c
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
```

The IOCTLS defined for the PmcHLnkBase driver are described below:

**IOCTL_PMC_HLNK_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:* PMC_HLNK_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 PMC_HLNK_BASE_DRIVER_DEVICE_INFO below.

```c
// Driver/Device information
typedef struct _PMC_HLNK_BASE_DRIVER_DEVICE_INFO {
    UCHAR DriverRev;
    UCHAR DesignId;
    UCHAR DesignRev;
    UCHAR MinorRev;
    UCHAR SwitchValue;
    UCHAR PllDeviceId;
    UCHAR InstanceNum;
} PMC_HLNK_BASE_DRIVER_DEVICE_INFO, *PPMC_HLNK_BASE_DRIVER_DEVICE_INFO;
```
IOCTL_PMC_HLNK_BASE_LOAD_PLL_DATA

**Function:** Writes to the internal registers of the PLL.

**Input:** PMC_HLNK_BASE_PLL_DATA structure

**Output:** None

**Notes:** The PLL internal register data is loaded into the PMC_HLNK_BASE_PLL_DATA structure in an array of 40 bytes. Appropriate values for this array can be derived from .jed files created by the CyberClock utility from Cypress Semiconductor. See below for the definition of PMC_HLNK_BASE_PLL_DATA.

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

typedef struct _PMC_HLNK_BASE_PLL_DATA {
    UCHAR Data[PLL_MESSAGE_SIZE];
} PMC_HLNK_BASE_PLL_DATA, *PPMC_HLNK_BASE_PLL_DATA;
```

IOCTL_PMC_HLNK_BASE_READ_PLL_DATA

**Function:** Returns the contents of the internal registers of the PLL.

**Input:** None

**Output:** PMC_HLNK_BASE_PLL_DATA structure

**Notes:** The PLL internal register data is read and inserted into the data structure in an array of 40 bytes. See the definition of PMC_HLNK_BASE_PLL_DATA above.

IOCTL_PMC_HLNK_BASE_GET_STATUS

**Function:** Returns the value of the status register and clears any latched bits

**Input:** None

**Output:** Status register value (unsigned int)

**Notes:** Returns the real-time values of the status bits and clears the bits in BASE_STAT_PLL_LATCH_MASK if they are set.

```c
/* Status bit definitions */
#define BASE_STAT_INT0_ACTV       0x00000001
#define BASE_STAT_INT1_ACTV       0x00000002
#define BASE_STAT_PLLREF_LCKD     0x00000040
#define BASE_STAT_HLCLK_LCKD      0x00000080
#define BASE_STAT_PLL_TX_FF_MT    0x00000100
#define BASE_STAT_PLL_TX_FF_FL    0x00000200
#define BASE_STAT_PLL_TX_FF_VLD   0x00000400
#define BASE_STAT_PLL_RX_FF_MT    0x00001000
#define BASE_STAT_PLL_RX_FF_FL    0x00002000
#define BASE_STAT_PLL_RX_FF_VLD   0x00004000
#define BASE_STAT_PLL_RDY         0x00010000
#define BASE_STAT_PLL_DONE        0x00020000
#define BASE_STAT_PLL_ERROR       0x00040000
#define BASE_STAT_CORE_REV_MASK   0x0FF00000
#define BASE_STAT_PLL_FIFO_MASK   (BASE_STAT_PLL_TX_FF_MT | BASE_STAT_PLL_TX_FF_FL | BASE_STAT_PLL_TX_FF_VLD |
                                BASE_STAT_PLL_RX_FF_MT | BASE_STAT_PLL_RX_FF_FL | BASE_STAT_PLL_RX_FF_VLD)
#define BASE_STAT_PLL_LATCH_MASK  (BASE_STAT_PLL_DONE | BASE_STAT_PLL_ERROR)
#define BASE_STAT_MASK            (BASE_STAT_INT0_ACTV | BASE_STAT_HLCLK_LCKD | BASE_STAT_PLL_FIFO_MASK |
                                BASE_STAT_INT1_ACTV | BASE_STAT_PLLREF_LCKD | BASE_STAT_PLL_LATCH_MASK |
                                BASE_STAT_PLL_RDY | BASE_STAT_CORE_REV_MASK)
```
The IOCTLs defined for the PmcHLnkChan driver are described below:

**IOCTL_PMC_HLNK_CHAN_GET_INFO**

*Function:* Returns the channel number driver revision as well as the board instance number, design ID, design revision and minor revision passed in from the base driver.

*Input:* None

*Output:* PMC_HLNK_CHAN_DRIVER_DEVICE_INFO structure

*Notes:* See the definition of PMC_HLNK_CHAN_DRIVER_DEVICE_INFO below.

```c
/* Driver/Device information */
typedef struct _HLNK_CHAN_DRIVER_DEVICE_INFO {
    unsigned char DriverRev; // Channel driver revision
    unsigned int InstanceNum; // Board instance number from base driver
    unsigned char Channel; // Channel number
    unsigned char DesignId; // From base driver
    unsigned char DesignRev; // From base driver
    unsigned char MinorRev; // From base driver
} HLNK_CHAN_DRIVER_DEVICE_INFO, *PHLNK_CHAN_DRIVER_DEVICE_INFO;
```

**IOCTL_PMC_HLNK_CHAN_SET_CONFIG**

*Function:* Sets the requested channel control configuration.

*Input:* PMC_HLNK_CHAN_CONFIG structure

*Output:* None

*Notes:* See below for the definitions of the structures used in this call.

```c
typedef struct _PMC_HLNK_CHAN_INTS {
    BOOLEAN TxAmtInt; // Transmit FIFO almost empty interrupt
    BOOLEAN RxAflInt; // Receive FIFO almost full interrupt
    BOOLEAN RxOvflInt; // Receive FIFO overflow interrupt
} PMC_HLNK_CHAN_INTS, *PPMC_HLNK_CHAN_INTS;
```

```c
// Channel DMA priority (use sparingly)
typedef enum _PMC_HLNK_DMA_PRMPT {
    PMC_HLNK_NONE, // No priority
    PMC_HLNK_READ, // Read DMA has priority
    PMC_HLNK_WRITE, // Write DMA has priority
    PMC_HLNK_RDWR // Read and Write DMA have priority
} PMC_HLNK_DMA_PRMPT, *PPMC_HLNK_DMA_PRMPT;
```

```c
/* Channel Configuration */
typedef struct _PMC_HLNK_CHAN_CONFIG {
    BOOLEAN TxEnable; // Enable HOTLink transmitter
    BOOLEAN RxEnable; // Enable HOTLink receiver
    BOOLEAN FfoTestEn; // Enables auto tx->rx FIFO transfer
    BOOLEAN IoTestEn; // Enables tx->rx I/O data transfer
    BOOLEAN TxOutEn; // Enable transmitter output
    BOOLEAN TxBitEn; // Built-in-test enable (sends test pattern)
    BOOLEAN TxLdEn; // Enables loading of test data
    BOOLEAN TxSndFrm; // Forces sending a data-frame without trigger
    BOOLEAN TtlCmdEn; // Enables TTL I/F to trigger sending a data-frame
    BOOLEAN RxInAsel; // Selects rx input '1'=External, '0'=Local Tx
    BOOLEAN RxBitEn; // Built-in-test enable (verifies test pattern)
    BOOLEAN RxReframe; // Manually initiate receiver data reframe
    BOOLEAN ForceRfrm; // Force reframe signal high
    PMC_HLNK_CHAN_INTS IntConfig; // Interrupt condition enables
    PMC_HLNK_DMA_PRMPT DmaPriority; // DMA preemption control
} PMC_HLNK_CHAN_CONFIG, *PPMC_HLNK_CHAN_CONFIG;
```
IOCTL_PMC_HLNK_CHAN_GET_CONFIG

**Function:** Returns the channel’s control configuration.

**Input:** None

**Output:** PMC_HLNK_CHAN_CONFIG structure

**Notes:** Returns the parameter values written in the previous call.

IOCTL_PMC_HLNK_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:** The latched bits in CHAN_STAT_LATCH_MASK will be cleared only if they are set when the status is read.

```c
/* 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_RX_DATA_RDY 0x00000100
#define CHAN_STAT_TX_DATA_READ 0x00000200
#define CHAN_STAT_RX_ACTIVE 0x00000400
#define CHAN_STAT_RX_SYNCHED 0x00000800
#define CHAN_STAT_RX_UDEF_CHAR 0x00001000
#define CHAN_STAT_RX_DISP_ERR 0x00002000
#define CHAN_STAT_LOC_INT 0x00004000
#define CHAN_STAT_INT_ACTIVE 0x00008000
#define CHAN_STAT_FIFO_MASK (CHAN_STAT_TX_FF_MT | CHAN_STAT_TX_FF_AMT | CHAN_STAT_TX_FF_VLD | CHAN_STAT_RX_FF_MT | CHAN_STAT_RX_FF_AFL | CHAN_STAT_RX_FF_VLD | CHAN_STAT_RX_FF_FL)
#define CHAN_STAT_LATCH_MASK (CHAN_STAT_RD_DMA_ERR | CHAN_STAT_RX_FRAME_DN | CHAN_STAT_TX_UNDRN_ERR | CHAN_STAT_RX_FRAME_DN | CHAN_STAT_TX_FRAME_ERR | CHAN_STAT_TX_FRAME_DN | CHAN_STAT_TX_COUNT_ERR | CHAN_STAT_RX_FRAME_DN | CHAN_STAT_RX_FRAME_DN | CHAN_STAT_TX_COUNT_ERR | CHAN_STAT_RX_SYM_ERR | CHAN_STAT_RX_DATA_RDY | CHAN_STAT_RX_COUNT_ERR | CHAN_STAT_RX_Disp_ERR | CHAN_STAT_RX_UDef_CHAR | CHAN_STAT_RX_DISP_ERR | CHAN_STAT_RX_UDef_CHAR)
#define CHAN_STAT_MASK (CHAN_STAT_WR_DMA_INT | CHAN_STAT_WR_DMA_RDY | CHAN_STAT_LOC_INT | CHAN_STAT_RD_DMA_INT | CHAN_STAT_RD_DMA_RDY | CHAN_STAT_FIFO_MASK | CHAN_STAT_RX_SYNCHED | CHAN_STAT_INT_ACTIVE | CHAN_STAT_RX_ACTIVE | CHAN_STAT_RX_Disp_ERR | CHAN_STAT_RX_UDef_Char | CHAN_STAT_LOC_INT | CHAN_STAT_FS_INT)
**IOCTL_PMC_HLNK_CHAN_SET_FIFO_LEVELS**

*Function:* Sets the transmitter almost empty and receiver almost full levels for the channel.

*Input:* PMC_HLNK_CHAN_FIFO_LEVELS structure

*Output:* None

*Notes:* These values are set to the default values ¼ FIFO and ¾ FIFO respectively when the driver initializes. The FIFO counts are compared to these levels to set the value of the CHAN_STAT_TX_FF_AMT and CHAN_STAT_RX_FF_AFL status bits. Also, if read and/or write DMA priority is selected, these levels are used to determine at what point DMA preemption for an input or output DMA channel will take effect. See the definition of PMC_HLNK_CHAN_FIFO_LEVELS below.

```c
typedef struct _PMC_HLNK_CHAN_FIFO_LEVELS {
    ULONG    AlmostFull;
    ULONG    AlmostEmpty;
} PMC_HLNK_CHAN_FIFO_LEVELS, *PPMC_HLNK_CHAN_FIFO_LEVELS;
```

**IOCTL_PMC_HLNK_CHAN_GET_FIFO_LEVELS**

*Function:* Returns the transmitter almost empty and receiver almost full levels for the channel.

*Input:* None

*Output:* PMC_HLNK_CHAN_FIFO_LEVELS structure

*Notes:* Returns the values set in the previous call.

**IOCTL_PMC_HLNK_CHAN_GET_FIFO_COUNTS**

*Function:* Returns the number of data words in the transmit and receive data FIFOs.

*Input:* None

*Output:* PMC_HLNK_CHAN_FIFO_COUNTS structure

*Notes:* There is one pipe-line latch for the transmit FIFO data and four for the receive FIFO data. These are counted in the FIFO counts. That means the transmit count can be a maximum of 32,769 32-bit words and the receive count can be a maximum of 32,772 32-bit words.

```c
typedef struct _PMC_HLNK_CHAN_FIFO_COUNTS {
    ULONG   TxCount;
    ULONG   RxCount;
} PMC_HLNK_CHAN_FIFO_COUNTS, *PPMC_HLNK_CHAN_FIFO_COUNTS;
```
IOCTL_PMC_HLNK_CHAN_RESET_FIFOS

*Function:* Resets one or both FIFOs for the referenced channel.
*Input:* PMC_HLNK_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 PMC_HLNK_CHAN_FIFO_SEL below.

```c
// Used for FIFO reset call
typedef enum PMC_HLNK_CHAN_FIFO_SEL {
    PMC_HLNK_TX,
    PMC_HLNK_RX,
    PMC_HLNK_BOTH
} PMC_HLNK_CHAN_FIFO_SEL, *PPMC_HLNK_CHAN_FIFO_SEL;
```

IOCTL_PMC_HLNK_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_PMC_HLNK_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_PMC_HLNK_CHAN_WRITE_RAM

*Function:* Writes a 32-bit-data-word to the format RAM.
*Input:* PMC_HLNK_CHAN_MEM_WORD_WRITE structure
*Output:* None
*Notes:* Used to write data-frame format information to the format RAM.

```c
typedef struct PMC_HLNK_CHAN_MEM_WORD_WRITE {
    unsigned int Address;
    unsigned int Data;
} PMC_HLNK_CHAN_MEM_WORD_WRITE, *PPMC_HLNK_CHAN_MEM_WORD_WRITE;
```
**IOCTL_PMC_HLNK_CHAN_READ_RAM**

*Function:* Reads a 32-bit frame format word from the format RAM.
*Input:* RAM word address (unsigned character)
*Output:* RAM format word (unsigned integer)
*Notes:* This call is used to test the RAM. In normal operation the format RAM is only read by the transmitter and receiver state-machines.

**IOCTL_PMC_HLNK_CHAN_GET_MSG_COUNTS**

*Function:* Reads and returns the byte counts from the last message sent/received.
*Input:* None
*Output:* PMC_HLNK_CHAN_MSG_COUNTS
*Notes:* See the definition of PMC_HLNK_CHAN_MSG_COUNTS below.

```c
typedef struct PMC_HLNK_CHAN_MSG_COUNTS {
    unsigned int TxMsgCount;
    unsigned int RxMsgCount;
} PMC_HLNK_CHAN_MSG_COUNTS, *PPMC_HLNK_CHAN_MSG_COUNTS;
```

**IOCTL_PMC_HLNK_CHAN_SET_TTL_CONFIG**

*Function:* Writes the channel TTL configuration parameters.
*Input:* PMC_HLNK_CHAN_TTL_CONFIG structure
*Output:* None
*Notes:* See the definition of PMC_HLNK_CHAN_TTL_CONFIG below.

```c
typedef struct PMC_HLNK_CHAN_TTL_CONFIG {
    BOOLEAN RxTtlEn; // Receive TTL data
    BOOLEAN TxTtlEn; // Load and send TTL data
    BOOLEAN TtlFifoTestEn; // Enables auto tx->rx FIFO transfer
    BOOLEAN TtlRxDnIntEn; // Enables the receiver frame done interrupt
} PMC_HLNK_CHAN_TTL_CONFIG, *PPMC_HLNK_CHAN_TTL_CONFIG;
```

**IOCTL_PMC_HLNK_CHAN_GET_TTL_CONFIG**

*Function:* Returns the channel’s TTL control configuration.
*Input:* None
*Output:* PMC_HLNK_CHAN_TTL_CONFIG structure
*Notes:* Returns the values set in the previous call. See the definition of PMC_HLNK_CHAN_TTL_CONFIG above.
**ioctl_pmclnk_chan_get_ttl_status**  
*Function*: Returns the channel’s TTL status register value.  
*Input*: None  
*Output*: Value of channel TTL status register (unsigned integer)  
*Notes*: The bits in CHAN_TTL_STAT_LAT_MASK will be cleared, if they are set when this call is made.

```
#define CHAN_TTL_STAT_TX_FF_MT 0x00000001
#define CHAN_TTL_STAT_TX_FF_AMT 0x00000002
#define CHAN_TTL_STAT_TX_FF_AFL 0x00000004
#define CHAN_TTL_STAT_TX_FF_VLD 0x00000008
#define CHAN_TTL_STAT_TX_FF_MT 0x00000010
#define CHAN_TTL_STAT_RX_FF_MT 0x00000020
#define CHAN_TTL_STAT_RX_FF_AFL 0x00000040
#define CHAN_TTL_STAT_RX_FF_VLD 0x00000080
#define CHAN_TTL_STAT_RX_BIT_ERR 0x00000100
#define CHAN_TTL_STAT_RX_FF_OVFL 0x00000200
#define CHAN_TTL_STAT_RX_DONE 0x00000400
#define CHAN_TTL_STAT_RX_TRIG_ERR 0x00000800
```

```
#define CHAN_TTL_STAT_TX_FF_MASK (CHAN_TTL_STAT_TX_FF_MT | CHAN_TTL_STAT_TX_FF_AMT | CHAN_TTL_STAT_TX_FF_AFL)
#define CHAN_TTL_STAT_RX_FF_MASK (CHAN_TTL_STAT_RX_FF_MT | CHAN_TTL_STAT_RX_FF_AFL)
#define CHAN_TTL_STAT_FF_MASK (CHAN_TTL_STAT_TX_FF_MASK | CHAN_TTL_STAT_RX_FF_MASK)
#define CHAN_TTL_STAT_LAT_MASK (CHAN_TTL_STAT_RX_BIT_ERR | CHAN_TTL_STAT_RX_DONE | CHAN_TTL_STAT_RX_FF_OVFL | CHAN_TTL_STAT_RX_TRIG_ERR)
#define CHAN_TTL_STAT_MASK (CHAN_TTL_STAT_FF_MASK | CHAN_TTL_STAT_LAT_MASK)
```

**ioctl_pmclnk_chan_get_ttl_fifo_counts**  
*Function*: Returns the number of data words in the transmitter and receiver TTL FIFOs.  
*Input*: None  
*Output*: PMC_HLNK_CHAN_FIFO_COUNTS structure  
*Notes*: There is one pipe-line latch for the transmitter and receiver FIFO. These are counted in the FIFO counts. That means the transmitter and receiver count can be a maximum of 4097 32-bit words.

```c
/* FIFO word counts */
typedef struct _PMC_HLNK_CHAN_FIFO_COUNTS {
    unsigned int TxCount;
    unsigned int RxCount;
} PMC_HLNK_CHAN_FIFO_COUNTS, *PPMC_HLNK_CHAN_FIFO_COUNTS;
```
**IOCTL_PMC_HLNK_CHAN_RESET_TTL_FIFOS**

*Function:* Resets one or both TTL FIFOs for the channel.

*Input:* PMC_HLNK_CHAN_FIFO_SEL enumeration type

*Output:* None

*Notes:* Resets the transmitter or receiver TTL FIFO or both depending on the input parameter selection.

```c
/* FIFO select (used by FIFO reset) */
typedef enum PMC_HLNK_CHAN_FIFO_SEL {
  PMC_HLNK_TX,
  PMC_HLNK_RX,
  PMC_HLNK_BOTH
} PMC_HLNK_CHAN_FIFO_SEL, *PPMC_HLNK_CHAN_FIFO_SEL;
```

**IOCTL_PMC_HLNK_CHAN_WRITE_TTL_FIFO**

*Function:* Writes a 32-bit data-word to the transmitter TTL FIFO.

*Input:* FIFO word (unsigned integer)

*Output:* None

*Notes:* Used to write data to the transmitter TTL FIFO.

**IOCTL_PMC_HLNK_CHAN_READ_TTL_FIFO**

*Function:* Reads and returns a 32-bit data word from the receiver TTL FIFO.

*Input:* None

*Output:* FIFO word (unsigned integer)

*Notes:* Used to read data from the receiver TTL FIFO.

**IOCTL_PMC_HLNK_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_PMC_HLNK_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_PMC_HLNK_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_PMC_HLNK_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_PMC_HLNK_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 PMC_HLNK_CHAN_ISR_STATUS.

```c
/* Interrupt status from ISR */
typedef struct _PMC_HLNK_CHAN_ISR_STAT {
    unsigned int HlStat; // HOTLink status read in the ISR
    unsigned int TtlStat; // TTL status read in the ISR
    BOOLEAN HlNew; // True if status has changed since the last get ISR status call
    BOOLEAN TtlNew; // True if TTL status has changed since the last get ISR status call
    BOOLEAN TimedOut; // True if interrupt wait time was exceeded
} PMC_HLNK_CHAN_ISR_STAT, *PPMC_HLNK_CHAN_ISR_STAT;
```
**IOCTL_PMC_HLNK_CHAN_READ_DMA_COUNTS**

**Function:** Returns the number of words transferred in the last input and output DMA.

**Input:** None

**Output:** PMC_HLNK_CHAN_DMA_COUNTS

**Notes:** This count will remain valid even if the board is reset. This allows the user to get information about a DMA transfer that was hung or failed to complete.

```c
typedef struct _PMC_HLNK_CHAN_DMA_COUNTS {
    unsigned int WriteCount;
    unsigned int ReadCount;
} PMC_HLNK_CHAN_DMA_COUNTS, *PPMC_HLNK_CHAN_DMA_COUNTS;
```

**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

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering’s sole responsibility shall be to repair, or at Dynamic Engineering’s sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering’s products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

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

All information provided is Copyright Dynamic Engineering.