hlnk_base
&
hlnk_chan

Linux Driver Documentation

Revision B
Corresponding Hardware: Revision C
10-2009-0103
Corresponding Firmware: Revision B2
hlk_base & hlk_chan
Linux Device Drivers for the
ccPMC-HOTLink-Kaon1 PMC Module
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.

©2018 by Dynamic Engineering
Other trademarks and registered trademarks are owned by their respective manufacturers.
Manual Revision B. Revised September 6, 2018
Table of Contents

Introduction ................................................................................................................................................. 4
Note ......................................................................................................................................................... 4
Driver Installation ................................................................................................................................. 4
Driver Startup .......................................................................................................................................... 5
IO Controls ................................................................................................................................................ 6
  IOCTL_HLNK_BASE_GET_INFO .............................................................................................................. 6
  IOCTL_HLNK_BASE_LOAD_PLL_DATA ................................................................................................. 6
  IOCTL_HLNK_BASE_READ_PLL_DATA ................................................................................................. 6
  IOCTL_HLNK_BASE_GET_STATUS ......................................................................................................... 7
  IOCTL_HLNK_CHAN_GET_INFO .......................................................................................................... 8
  IOCTL_HLNK_CHAN_SET_CONFIG ....................................................................................................... 8
  IOCTL_HLNK_CHAN_GET_CONFIG ...................................................................................................... 9
  IOCTL_HLNK_CHAN_SET_FIFO_LEVELS ............................................................................................. 9
  IOCTL_HLNK_CHAN_GET_FIFO_LEVELS ............................................................................................. 10
  IOCTL_HLNK_CHAN_GET_FIFO_COUNTS ............................................................................................ 10
  IOCTL_HLNK_CHAN_RESET_FIFO ....................................................................................................... 10
  IOCTL_HLNK_CHAN_WRITE_FIFO ....................................................................................................... 11
  IOCTL_HLNK_CHAN_READ_FIFO .......................................................................................................... 11
  IOCTL_HLNK_CHAN_WRITE_RAM ........................................................................................................ 11
  IOCTL_HLNK_CHAN_READ_RAM .......................................................................................................... 11
  IOCTL_HLNK_CHAN_GET_MSG_COUNTS .............................................................................................. 11
  IOCTL_HLNK_CHAN_SET_TTL_CONFIG ............................................................................................... 12
  IOCTL_HLNK_CHAN_GET_TTL_CONFIG ............................................................................................... 12
  IOCTL_HLNK_CHAN_GET_TTL_STATUS ............................................................................................... 12
  IOCTL_HLNK_CHAN_GET_TTL_FIFO_COUNTS .................................................................................... 13
  IOCTL_HLNK_CHAN_RESET_TTL_FIFO ............................................................................................... 13
  IOCTL_HLNK_CHAN_WRITE_TTL_FIFO ............................................................................................... 13
  IOCTL_HLNK_CHAN_READ_TTL_FIFO ............................................................................................... 13
  IOCTL_HLNK_CHAN_WAIT_ON_INTERRUPT ....................................................................................... 14
  IOCTL_HLNK_CHAN_ENABLE_INTERRUPT .......................................................................................... 14
  IOCTL_HLNK_CHAN_DISABLE_INTERRUPT ....................................................................................... 14
  IOCTL_HLNK_CHAN_FORCE_INTERRUPT ............................................................................................ 14
  IOCTL_HLNK_CHAN_GET_ISR_STATUS ............................................................................................... 15
  IOCTL_HLNK_CHAN_READ_DMA_COUNT ............................................................................................ 15
Write ......................................................................................................................................................... 16
Read ......................................................................................................................................................... 16
Warranty and Repair ............................................................................................................................. 17
Service Policy ........................................................................................................................................ 17
  Out of Warranty Repairs .................................................................................................................. 17
For Service Contact: ............................................................................................................................ 17
**Introduction**

The hlnk_base and hlnk_chan drivers are Linux device drivers for the ccPMC-HOTLink-Kaon1 from Dynamic Engineering. The HOTLink board has a Spartan6-100 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for two HOTLink channels. There is also a programmable PLL with three clock outputs, one for the HOTLink reference frequency (16.777216 MHz), one for the TTL 4x reference frequency (27.52512 MHz) and the last for sampling the input TTL signal, which is 5x the oscillator frequency (147.456 MHz). Each channel has a 32k x 32-bit receive FIFO and a 32k x 32-bit transmit FIFO for the HOTLink interface and two 4k x 32-bit FIFOs for the input and output TTL interface lines.

When the hlnk_base module is loaded, it interfaces with the PCI bus sub-system to acquire the memory and interrupt resources for each HOTLink board installed. An hlnk_bus is created for each device and two channel devices are allocated. The interrupt is assigned and the address space partitioned for the two channel devices. When the hlnk_chan driver is installed, it probes the hlnk bus and finds and initializes the two channel devices for each board. It allocates read and write list memory to hold the DMA page descriptors that are used by the hardware to perform bus-master scatter-gather DMA.

**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 HOTLink user manual (also referred to as the hardware manual). The HOTLink base and channel drivers were developed on Linux kernel version 4.9.0-6. If you are using a different version, some modification of the source code might be required.

**Driver Installation**

The source files and Makefiles for the drivers and test application are supplied in the driver archive file PmcHOTLinkKaon_9_6_18.zip. Extract and copy the directory structure to the computer where the driver is to be built. From the top-level directory type “make” to build the object files then type “make install” to copy the files to the target location (must be root for this to succeed) (/lib/modules/$(VERSION)/kernel/drivers/char/hlnk for the driver and /usr/local/bin/ for the test app). After installation, you can type “make clean” to remove object files and executables.

A load_hlnk script is provided that will load the base driver, parse the /proc/devices file for the device’s major number, count the number of entries in the /sys/bus/hlnk/devices/ directory to determine the number of boards installed, create the required number of /dev/hlnk_base_x (where x is the zero based board number) device nodes, load the channel driver, find that major number and create the required number of /dev/hlnk_chan_y device nodes as well.
The Application Program Interface (API) for the drivers and relevant structures and bit defines for the control/status registers on the ccPMC-HOTLink-Kaon1 are defined in the C header files hlnk_base_api.h and hlnk_chan_api.h. The user_app source code will provide examples of how to use the driver calls to control the hardware.

**Driver Startup**

Install the hardware and boot the computer. After the drivers have been installed run the load_hlnk script to start the drivers and create the device interface nodes.

Handles can be opened to a specific board by using the open() function call and passing in the appropriate device names.

Below is example code for opening handles for device `dev_num`.

```c
#define INPUT_SIZE 80

HANDLE   hhlnk_base;
HANDLE   hhlnk_chan[HLNK_BASE_NUM_CHANNELS];

doi
printf("Enter target board number (starting with zero): \n");
restore_term();
scanf("%d", &dev_num);
init_term();
if (dev_num < 0 || dev_num > NUM_HLNK_DEVICES) {
    printf("Target board number %d out of range!\n", dev_num);
    return 1;
}while (dev_num < 0 || dev_num > NUM_HLNK_DEVICES);

sprintf(Name, "/dev/hlnk_base_%d", dev_num);
hhlnk_base = open(Name, O_RDWR);
if (hhlnk_base < 2) {
    printf("Failed to open!\n", Name);
    restore_term();
    return 1;
}

chan_num = dev_num * HLNK_BASE_NUM_CHANNELS 

for (i = 0; i < HLNK_BASE_NUM_CHANNELS; i++) {
    sprintf(Name, "/dev/hlnk_chan_%d", chan_num + i);
    hhlnk_chan[i] = open(Name, O_RDWR);
    if (hhlnk_chan[i] < 2) {
        printf("Failed to open!\n", Name);
        restore_term();
        return 1;
    }
}
menu();
```
IO Controls

The driver uses ioctl() calls to configure the device and obtain status. The parameters passed to the ioctl() function include the handle obtained from the open() call, an integer command number defined in the API header files and an optional parameter used to pass data in and/or out of the device. The ioctl commands defined for the PMC-HOTLink are listed below.

The ioctl() calls defined for the hlnk_base driver are described below:

IOCTL_HLNK_BASE_GET_INFO

Function: Returns the Driver revision, Design ID, Design revision, Switch value, Instance number, and PLL device ID.

Input: None

Output: HLNK_BASE_DRIVER_DEVICE_INFO structure

Notes: Switch value is the configuration of the on-board dip-switch that has been set by the user (see the board silk screen for bit position and polarity). The PLL device ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may alternatively be 0x6A. See below for the definition of HLNK_BASE_DRIVER_DEVICE_INFO.

default struct _HLNK_BASE_DRIVER_DEVICE_INFO {
    unsigned char DriverRev;
    unsigned char DesignId;
    unsigned char DesignRev;
    unsigned char MinorRev;
    unsigned char SwitchValue;
    unsigned char PllDeviceId;
    unsigned int InstanceNum;
} HLNK_BASE_DRIVER_DEVICE_INFO, *PHLNK_BASE_DRIVER_DEVICE_INFO;

IOCTL_HLNK_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.

Input: HLNK_BASE_PLL_DATA structure

Output: None

Notes: The PLL internal register data is loaded into the 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.

default struct _HLNK_BASE_PLL_DATA {
    unsigned char Data[PLL_MESSAGE_SIZE];
} HLNK_BASE_PLL_DATA, *PHLNK_BASE_PLL_DATA;

IOCTL_HLNK_BASE_READ_PLL_DATA

Function: Returns the contents of the PLL’s internal registers

Input: None

Output: HLNK_BASE_PLL_DATA structure

Notes: The register data is output in the HLNK_BASE_PLL_DATA structure in an array of 40 bytes.
**FUNCTION: IOCTL_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_HLCLK_LCKD 0x00000040
#define BASE_STAT_PLLREF_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_PLLREF_LCKD | BASE_STAT_PLL_LATCH_MASK | BASE_STAT_PLL_RDY | BASE_STAT_CORE_REV_MASK)
```
The ioctl() calls defined for the hlnk_chan driver are described below:

**IOCTL_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:* HLNK_CHAN_DRIVER_DEVICE_INFO structure

*Notes:* See the definition of 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_HLNK_CHAN_SET_CONFIG**

*Function:* Writes the channel configuration parameters.

*Input:* HLNK_CHAN_CONFIG structure

*Output:* None

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

```c
/* Channel Interrupt Enables */
typedef struct _HLNK_CHAN_INTS {
    BOOLEAN TxAmtInt;                 // Transmit FIFO almost empty interrupt
    BOOLEAN RxAflInt;                 // Receive FIFO almost full interrupt
    BOOLEAN RxOvflInt;                // Receive FIFO overflow interrupt
} HLNK_CHAN_INTS, *PHLNK_CHAN_INTS;

/* Channel DMA priority */
typedef enum _HLNK_DMA_PRMPT {
    HLNK_NONE,          // No priority
    HLNK_READ,          // Read DMA has priority
    HLNK_WRITE,         // Write DMA has priority
    HLNK_RDWR           // Read and Write DMA have priority
} HLNK_DMA_PRMPT, *PHLNK_DMA_PRMPT;

/* Channel Configuration */
typedef struct _HLNK_CHAN_CONFIG {
    BOOLEAN TxEnable;              // Enable HOTLink transmitter
    BOOLEAN RxEnable;              // Enable HOTLink receiver
    BOOLEAN FifoTestEn;            // 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 (verifies test pattern)
    BOOLEAN TxSndFrm;              // Forces sending a data-frame without trigger
    BOOLEAN TtlCmndEn;             // 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
    HLNK_CHAN_INTS IntConfig;      // Interrupt condition enables
    HLNK_DMA_PRMPT DmaPriority;    // DMA preemption control
} HLNK_CHAN_CONFIG, *PHLNK_CHAN_CONFIG;
```
**IOCTL_HLNK_CHAN_GET_CONFIG**

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

*Input:* None

*Output:* HLNK_CHAN_CONFIG structure

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

**IOCTL_HLNK_CHAN_GET_STATUS**

*Function:* Returns the channel’s status bit values and clears the latched bits.

*Input:* None

*Output:* Value of channel status register (unsigned integer)

*Notes:* The bits in CHAN_STAT_LATCH_MASK will be cleared 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_VLD 0x00000040
#define CHAN_STAT_RX_FF_FL 0x00000080
#define CHAN_STAT_TX_AMT_INT 0x00001000
#define CHAN_STAT_RX_AFL_INT 0x00002000
#define CHAN_STAT_RX_OVFL 0x00004000
#define CHAN_STAT_RX_SYM_ERR 0x00008000
#define CHAN_STAT_WR_DMA_INT 0x00010000
#define CHAN_STAT_RD_DMA_INT 0x00020000
#define CHAN_STAT_WR_DMA_ERR 0x00040000
#define CHAN_STAT_RD_DMA_ERR 0x00080000
#define CHAN_STAT_WR_DMA_RDY 0x00100000
#define CHAN_STAT_RD_DMA_RDY 0x00200000
#define CHAN_STAT_RX_DATA_RDY 0x00400000
#define CHAN_STAT_TX_DATA_READ 0x00800000
#define CHAN_STAT_RD_DMA_RDY 0x01000000
#define CHAN_STAT_RX_FRAME_DN 0x02000000
#define CHAN_STAT_RX_SYNCHED 0x04000000
#define CHAN_STAT_RX_UDEF_CHAR 0x08000000
#define CHAN_STAT_RX_DISP_ERR 0x10000000
#define CHAN_STAT_RD_DMA_INT 0x20000000
#define CHAN_STAT_RD_DMA_RDY 0x40000000
#define CHAN_STAT_RD_DMA_ERR 0x80000000
#define CHAN_STAT_FIFO_MASK (CHAN_STAT_TX_FF_MT | CHAN_STAT_TX_FF_FL | CHAN_STAT_TX_FF_AMT |
| CHAN_STAT_TX_FF_VLD | CHAN_STAT_RX_FF_MT | CHAN_STAT_RX_FF_AFL |
| CHAN_STAT_TX_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_ERR | CHAN_STAT_RX_FRAME_DN | CHAN_STAT_RX_UNDRN_ERR |
| CHAN_STAT_RX_FRAME_ERR | CHAN_STAT_RX_DATA_RDY | CHAN_STAT_RX_FRAME_DN |
| CHAN_STAT_TX_AMT_INT | CHAN_STAT_TX_FRAME_DN | CHAN_STAT_TX_FRAME_DN |
| CHAN_STAT_TX_UDEF_CHAR | CHAN_STAT_TX_FRAME_DN |
| CHAN_STAT_TX_UDEF_CHAR | CHAN_STAT_RX_FRAME_DN |
| CHAN_STAT_RX_UDEF_CHAR | CHAN_STAT_RX_FRAME_DN |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR | CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR | CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDEF_CHAR |
| CHAN_STAT_RX_UDE
IOCTL_HLNK_CHAN_SET_FIFO_LEVELS
Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: HLNK_CHAN_FIFO_LEVELS structure
Output: None
Notes: These values are initialized to the default values ¼ transmit FIFO size and ¾ receive FIFO size respectively when the driver initializes. The FIFO counts are compared to these levels to determine 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.

```c
/* FIFO programmable TX almost empty RX almost full levels */
typedef struct _HLNK_CHAN_FIFO_LEVELS {
    unsigned int AlmostFull;
    unsigned int AlmostEmpty;
} HLNK_CHAN_FIFO_LEVELS, *PHLNK_CHAN_FIFO_LEVELS;
```

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

IOCTL_HLNK_CHAN_GET_FIFO_COUNTS
Function: Returns the number of data words in the transmitter and receiver FIFOs.
Input: None
Output: 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
/* FIFO word counts */
typedef struct _HLNK_CHAN_FIFO_COUNTS {
    unsigned int TxCount;
    unsigned int RxCount;
} HLNK_CHAN_FIFO_COUNTS, *PHLNK_CHAN_FIFO_COUNTS;
```

IOCTL_HLNK_CHAN_RESET_FIFOS
Function: Resets one or both of the channel's HOTLink FIFOs.
Input: HLNK_CHAN_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 HLNK_CHAN_FIFO_SEL below.

```c
/* FIFO select (used by FIFO reset) */
typedef enum _HLNK_CHAN_FIFO_SEL {
    HLNK_TX,
    HLNK_RX,
    HLNK_BOTH
} HLNK_CHAN_FIFO_SEL, *PHLNK_CHAN_FIFO_SEL;
```
**IOCTL_HLNK_CHAN_WRITE_FIFO**  
*Function:* Writes a 32-bit data-word to the transmit FIFO.  
*Input:* FIFO word (unsigned integer)  
*Output:* None  
*Notes:* Used to make single-word accesses to the transmit FIFO instead of using DMA.

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

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

```c
typedef struct _HLNK_CHAN_MEM_WORD_WRITE {
    unsigned int  Address;
    unsigned int  Data;
} HLNK_CHAN_MEM_WORD_WRITE, *PHLNK_CHAN_MEM_WORD_WRITE;
```

**IOCTL_HLNK_CHAN_READ_RAM**  
*Function:* Reads a 32-bit frame format word from the format RAM.  
*Input:* RAM word address (unsigned integer)  
*Output:* RAM format word (unsigned integer)  
*Notes:* A union is used to contain the input and output parameters. Used to read format information from the specified address in the format RAM.

```c
typedef union _HLNK_CHAN_MEM_WORD_READ {
    unsigned int address;
    unsigned int data;
} HLNK_CHAN_MEM_WORD_READ, *PHLNK_CHAN_MEM_WORD_READ;
```

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

```c
typedef struct _HLNK_CHAN_MSG_COUNTS {
    unsigned int  TxMsgCount;
    unsigned int  RxMsgCount;
} HLNK_CHAN_MSG_COUNTS, *PHLNK_CHAN_MSG_COUNTS;
```
IOCTL_HLNK_CHAN_SET_TTL_CONFIG

*Function:* Writes the channel TTL configuration parameters.

*Input:* HLNK_CHAN_TTL_CONFIG structure

*Output:* None

*Notes:* See the definition of HLNK_CHAN_TTL_CONFIG below.

```c
typedef struct _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 RX done interrupt
} HLNK_CHAN_TTL_CONFIG, *PHLNK_CHAN_TTL_CONFIG;
```

IOCTL_HLNK_CHAN_GET_TTL_CONFIG

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

*Input:* None

*Output:* HLNK_CHAN_TTL_CONFIG structure

*Notes:* Returns the values set in the previous call. See the definition of HLNK_CHAN_TTL_CONFIG above.

IOCTL_HLNK_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 were set when this call was made.

```c
#define CHAN_TTL_STAT_TX_FF_MT 0x00000001
#define CHAN_TTL_STAT_TX_FF_AMT 0x00000002
#define CHAN_TTL_STAT_TX_FF_FL 0x00000004
#define CHAN_TTL_STAT_TX_FF_VLD 0x00000008
#define CHAN_TTL_STAT_RX_FF_AFL 0x00000020
#define CHAN_TTL_STAT_RX_FF_MT 0x00000010
#define CHAN_TTL_STAT_RX_FF_FL 0x00000040
#define CHAN_TTL_STAT_RX_FF_VLD 0x00000080
#define CHAN_TTL_STAT_TX_FF_MASK (CHAN_TTL_STAT_TX_FF_FL | CHAN_TTL_STAT_TX_FF_VLD | CHAN_TTL_STAT_TX_FF_MT | CHAN_TTL_STAT_TX_FF_AFL)
#define CHAN_TTL_STAT_RX_FF_MASK (CHAN_TTL_STAT_RX_FF_FL | CHAN_TTL_STAT_RX_FF_VLD | 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_HLNK_CHAN_GET_TTL_FIFO_COUNTS
Function: Returns the number of data words in the transmitter and receiver TTL FIFOs.
Input: None
Output: HLNK_CHAN_FIFO_COUNCTS 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.

/* FIFO word counts */
typedef struct _HLNK_CHAN_FIFO_COUNTS {
    unsigned int TxCount;
    unsigned int RxCount;
} HLNK_CHAN_FIFO_COUNTS, *PHLNK_CHAN_FIFO_COUNTS;

IOCTL_HLNK_CHAN_RESET_TTL_FIFOS
Function: Resets one or both TTL FIFOs for the channel.
Input: HLNK_CHAN_FIFO_SEL enumeration type
Output: None
Notes: Resets the transmitter or receiver TTL FIFO or both depending on the input parameter selection.

/* FIFO select (used by FIFO reset) */
typedef enum _HLNK_CHAN_FIFO_SEL {
    HLNK_TX,
    HLNK_RX,
    HLNK_BOTH
} HLNK_CHAN_FIFO_SEL, *PHLNK_CHAN_FIFO_SEL;

IOCTL_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_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_HLNK_CHAN_WAIT_ON_INTERRUPT
*Function:* Inserts the calling process into the interrupt wait queue until an interrupt occurs.
*Input:* Time-out value in jiffies (unsigned integer)
*Output:* None
*Notes:* This call is used to implement a user defined interrupt service routine. It will return when an interrupt occurs or when the delay time specified expires. If the delay is set to zero, the call will wait indefinitely. The delay time is dependent on the platform setting for jiffy, which could be anything from 10 milliseconds to less than 1 millisecond. The DMA interrupts do not use this mechanism; they are controlled automatically by the driver.

IOCTL_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 to re-enable interrupts after an interrupt occurs.

IOCTL_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_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 if the channel master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.
**IOCTL_HLNK_CHAN_GET_ISR_STATUS**

*Function:* Returns the interrupt status that was read in the ISR from the last user interrupt.

*Input:* None

*Output:* HLNK_CHAN_ISR_STAT structure

*Notes:* The HiStat and TtlStat fields are the status values that were read in the last interrupt service routine that serviced an enabled user interrupt. The HiNew and TtlNew fields are true if their respective interrupts occurred and updated the values since they were last read. The TimedOut field of the structure will be true if a time-out value was set in IOCTL_HLNK_CHAN_WAIT_ON_INTERRUPT and was exceeded. The interrupts that deal with the DMA transfers do not affect these values.

```c
typedef struct _HLNK_CHAN_ISR_STAT {
    unsigned int   HiStat; // HOTLink status read in the ISR
    unsigned int   TtlStat; // TTL status read in the ISR
    BOOLEAN        HiNew;  // 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
} HLNK_CHAN_ISR_STAT, *PHLNK_CHAN_ISR_STAT;
```

**IOCTL_HLNK_CHAN_READ_DMA_COUNTS**

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

*Input:* None

*Output:* HLNK_CHAN_DMA_COUNTS

*Notes:* These counts 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 _HLNK_CHAN_DMA_COUNTS {
    unsigned int   WriteCount;
    unsigned int   ReadCount;
} HLNK_CHAN_DMA_COUNTS, *PHLNK_CHAN_DMA_COUNTS;
```
Write

HOTLink transmit data is written to the device using the write command. A handle to the device, a pointer to a pre-allocated buffer that contains the data to write and an unsigned integer that represents the number of bytes of data to write are passed to the write call. The driver will obtain physical addresses to the pages containing the data and will set-up a list of page descriptors in its list memory. The driver writes the physical address of the first list entry to the device’s Write DMA pointer register. This triggers the hardware to perform a bus-master scatter-gather DMA to the device to transfer the data.

Read

HOTLink received data is read from the device using the read command. A handle to the device, a pointer to a pre-allocated buffer that will contain the data that is read from the device and an unsigned integer that represents the number of bytes of data to read are passed to the read call. The driver will obtain physical addresses to the buffer memory pages and will set-up a list of page descriptors in its list memory. The driver will write the physical address of the first list entry to the device’s Read DMA pointer register. This triggers the hardware to perform a bus-master scatter-gather DMA from the device to transfer the data.
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 St. Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax
support@dyneng.com

All information provided is Copyright Dynamic Engineering.