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

hlnk_base
&
hlnk_chan

Linux Driver Documentation

Revision A
Corresponding Hardware: Revision A
10-2009-0101
Corresponding Firmware: Revision A

hlnk_base & hlnk_chan
Linux Device Drivers for the
ccPMC-HOTLink PMC Module
6-Channel HOTLink Interface

Dynamic Engineering
150 DuBois St. Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 FAX

Manual Revision A. Revised April 17, 2009

Table of Contents

Introduction

The hlnk_base and hlnk_chan drivers are Linux device drivers for the ccPMC-HOTLink from Dynamic Engineering. The HOTLink board has a Spartan3-4000 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for six HOTLink channels. There is also a programmable PLL with two clock outputs, one for the HOTLink reference frequency and one for the RS-485 16x reference frequency. Each channel has an 4k x 32-bit receive FIFO and a 2k x 32-bit transmit FIFO for the HOTLink interface and two 1k x 32-bit FIFOs for the two RS-485 bidirectional interface lines.

When the hlnk_base module is installed, it interfaces with the PCI bus sub-system to acquire the memory and interrupt resources for each device installed. An hlnk bus is created for each device and six channel devices are allocated. The interrupt is assigned and the address space partitioned for the six channel devices. When the hlnk_chan driver is installed, it probes the hlnk bus and finds and initializes the six 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 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 2.6.18. 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 hot_link.tar.bz2. 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/misc/ 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 bit defines for the control/status registers on the PMC-HOTLink 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`.

```
#typedef long  HANDLE
#define  INPUT_SIZE  80

HANDLE   hhlnk_base;
HANDLE   hhlnk_chan[HLNK_BASE_NUM_CHANNELS];

char  Name[INPUT_SIZE];
int   i;
int   dev_num;
int   chan_num;

do
{
   printf("\nEnter target board number (starting with zero): \n");
   scanf("%d", &dev_num);
   if(dev_num < 0 || dev_num > NUM_HLNK_DEVICES)
      printf("\nTarget board number %d out of range!\n", dev_num);
}
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("\n%s FAILED to open!\n", Name);
   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("\n%s FAILED to open!\n", Name);
      return 1;
   }
}
```

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.

IOCTL_HLNK_BASE_GET_INFO

Function: Returns the Driver version, Xilinx 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 hlnk_base_api.h for the definition of HLNK_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.

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.

IOCTL_HLNK_CHAN_GET_INFO

Function: Returns the driver version and instance number of the referenced channel.
Input: None
Output: HLNK_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of HLNK_CHAN_DRIVER_DEVICE_INFO in the hlnk_chan_api.h header file.

IOCTL_HLNK_CHAN_SET_CONFIG

Function: Writes a configuration value to the channel control register.
Input: Value of channel control register (unsigned long integer)
Output: None
Notes: See hlnk_chan_api.h for the relevant channel control bit definitions. Only the bits in CHAN_CNTRL_MASK can be controlled by this call.

IOCTL_HLNK_CHAN_GET_CONFIG

Function: Returns the channel's control configuration.
Input: None
Output: Value of the channel control register (unsigned long integer)
Notes: Returns the values of the bits in CHAN_CNTRL_READ_MASK.

IOCTL_HLNK_CHAN_GET_STATUS

Function: Returns the channel's status value and clears the latched bits.
Input: None
Output: Value of channel status register (unsigned long integer)
Notes: The latched bits in CHAN_STAT_LATCH_MASK will be cleared if they are set when the status is read.

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 _ transmit FIFO size and _ 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 the control bits CHAN_CNTRL_URGNT_IN_EN and/or CHAN_CNTRL_URGNT_OUT_EN are set, these levels are used to determine when to give priority to an input or output DMA channel.

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:

IOCTL_HLNK_CHAN_GET_FIFO_COUNTS
Function: Returns the number of data words in the transmit and receive 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 2049 32-bit words and the receive count can be a maximum of 4100 32-bit words.

IOCTL_HLNK_CHAN_RESET_FIFOS
Function: Resets one or both HOTLink FIFOs for the channel
Input: HLNK_CHAN_FIFO_SEL enumeration type
Output: None
Notes: Resets the transmit or receive FIFO or both depending on the input parameter selection.

IOCTL_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_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_HLNK_CHAN_SET_485_CONFIG
Function: Writes a configuration value to the channel RS-485 control register.
Input: Value of channel RS-485 control register (unsigned long integer)
Output: None
Notes: See hlnk_chan_api.h for the relevant channel RS-485 control bit definitions. Only the bits in CHAN_485_CNTRL_MASK can be controlled by this call.

IOCTL_HLNK_CHAN_GET_485_CONFIG
Function: Returns the channel's RS-485 control configuration.
Input: None
Output: Value of the channel RS-485 control register (unsigned long integer)
Notes: Returns the values of the bits in CHAN_485_CNTRL_MASK.

IOCTL_HLNK_CHAN_GET_485_STATUS
Function: Returns the channel's RS-485 status register value.
Input: None
Output: Value of channel RS-485 status register (unsigned long integer)
Notes:

IOCTL_HLNK_CHAN_RESET_485_FIFOS
Function: Resets one or both RS-485 FIFOs for the channel
Input: HLNK_CHAN_485_FIFO_SEL enumeration type
Output: None
Notes: Resets the RS-485A or RS-485B FIFO or both depending on the input parameter selection.

IOCTL_HLNK_CHAN_WRITE_485A_FIFO
Function: Writes a 32-bit data-word to the RS-485A FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to write data to the RS-485A FIFO.

IOCTL_HLNK_CHAN_READ_485A_FIFO
Function: Returns a 32-bit data word from the RS-485A FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to read data from the RS-485A FIFO.

IOCTL_HLNK_CHAN_WRITE_485B_FIFO
Function: Writes a 32-bit data-word to the RS-485B FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: Used to write data to the RS-485B FIFO.

IOCTL_HLNK_CHAN_READ_485B_FIFO
Function: Returns a 32-bit data word from the RS-485B FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes: Used to read data from the RS-485B 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 long 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 as long as 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 read in the ISR from the last user interrupt.
Input: None
Output: HLNK_CHAN_ISR_STAT structure
Notes: The Status field is the status that was read in the last interrupt service routine that serviced an enabled user interrupt. The TimedOut field of the structure will be true if the time-out value set in IOCTL_HLNK_CHAN_WAIT_ON_INTERRUPT was exceeded. The interrupts that deal with the DMA transfers do not affect this value.

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 long 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 physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA 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 read and an unsigned long 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 physical address of the first list entry is written to the board, which performs a bus-master scatter-gather DMA 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 merchandisability 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. The current minimum repair charge is $125. 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