DYNAMIC ENGINEERING
150 DuBois St. Suite 3, Santa Cruz, Ca 95060
831-457-8891 Fax: 831-457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PB3Hw4

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision A
10-2007-0401
Corresponding Firmware: Revision A

PB3Hw4 WDM driver for the
PMC-BiSerial-III-TRANS-HW4
Bi-Directional Serial Data Interface
PMC Module

Dynamic Engineering
150 DuBois, Suite 3
Santa Cruz, CA 95060
831-457-8891
FAX: 831-457-4793

Manual Revision A. Revised September 26, 2007

Table of Contents

Introduction

The PB3Hw4 driver is a Win32 driver model (WDM) device driver for the PMC-BiSerial-III-TRANS-HW4 from Dynamic Engineering. The PMC-BiSerial-III-TRANS-HW4 board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, Dual-Port RAM and protocol control and status for 8 channels. There is also a programmable PLL with one clock output that can be selected as a clock reference instead of the oscillator. Each channel has a 2k-byte dual-port RAM for data transmission and reception.

When the PMC-BiSerial-III-TRANS-HW4 is recognized by the PCI bus configuration utility it will start the PB3Hw4 driver to allow communication with the device. IO Control calls (IOCTLs) are used to configure the board and read status. Read and Write calls are used to move blocks of data in and out of the device.

Note

This documentation will provide information about all calls made to the driver, and how the driver interacts with the device for each of these calls. For more detailed information on the hardware implementation, refer to the PMC-BiSerial-III-TRANS-HW4 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PB3Hw4.sys, PB3Hw4.inf, DDPB3Hw4.h, PB3Hw4GUID.h, Hw4Test.exe, and Hw4Test source files.

DDPB3Hw4.h is a C header file that defines the Application Program Interface (API) to the driver. PB3Hw4GUID.h is a C header files that defines the device interface identifier for the PB3Hw4 driver. These files are required at compile time by any application that wishes to interface with the driver, but they are not needed for driver installation.

The PB3Hw4Test.exe file is a sample Win32 console application that makes calls into the PB3Hw4 driver to test each driver call without actually writing any application code. It is not required during the driver installation.

To run PB3Hw4Test.exe, open a command prompt console window and type a command. Type ***PB3Hw4Test -d0 -?*** to display a list of commands (the PB3Hw4Test.exe file must be in the directory that the window is referencing). The commands are all of the form ***PB3Hw4Test -dn -im*** where ***n*** and ***m*** are the device number and PB3Hw4 driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

Windows 2000 Installation

Copy PB3Hw4.inf and PB3Hw4.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III-TRANS-HW4 hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

- Select ***Next***.
- Select ***Search for a suitable driver for my device.***
- Select ***Next***.
- Insert the disk prepared above in the desired drive.
- Select the appropriate drive e.g. ***Floppy disk drives***.
- Select ***Next***.
- The wizard should find the PB3Hw4.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

Windows XP Installation

Copy PB3Hw4.inf and PB3Hw4.sys to a floppy disk, or CD if preferred.

With the PMC-BiSerial-III-TRANS-HW4 Hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

- Insert the disk prepared above in the appropriate drive.
- Select ***No when asked to connect to Windows Update.***
- Select ***Next***.
- Select ***Install the software automatically.***
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

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 a globally unique identifier (GUID), which is defined in PB3Hw4GUID.h.

Below is example code for opening a handle for device *devNum*.

```
// Maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256

// Handle to device objects
HANDLE hPB3Hw4                          =  INVALID_HANDLE_VALUE;

// PB3Hw4 device number
ULONG                               devNum;

// Return status from command
LONG                                status;

// Handle to device interface information structure
HDEVINFO                            hDeviceInfo;

// The actual symbolic link name to use in the CreateFile() call
CHAR                                deviceName[MAX_DEVICE_NAME];

// Size of buffer reguired to get the symbolic link name
DWORD                               requiredSize;

// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA            interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;

hDeviceInfo = SetupDiGetClassDevs(
                  (LPGUID)&GUID_DEVINTERFACE_PB3_HW4,
                           NULL,
                           NULL,
                           DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}
```

```
interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                        (LPGUID)&GUID_DEVINTERFACE_PB3_HW4,
                                devNum,
                               &interfaceData))
{
   status = GetLastError();
   if(status == ERROR_NO_MORE_ITEMS)
   {
      printf("**Error: couldn't find device(no more items), (%d)\n", devNum);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
   else
   {
      printf("**Error: couldn't enum device, (%d)\n", status);
      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Found our device-get the details data to obtain the symbolic link name
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                     NULL,
                                     0,
                                    &requiredSize,
                                     NULL))
{
   if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
   {
      printf("**Error: couldn't get interface detail, (%d)\n",
              GetLastError());

      SetupDiDestroyDeviceInfoList(hDeviceInfo);
      exit(-1);
   }
}

// Allocate a buffer to get detail
pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
if(pDeviceDetail == NULL)
{
   printf("**Error: couldn't allocate interface detail\n");
   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   exit(-1);
}

pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
```

```
// Get the detail info
if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                    &interfaceData,
                                     pDeviceDetail,
                                     requiredSize,
                                     NULL,
                                     NULL))
{
   printf("**Error: couldn't get interface detail(2), (%d)\n",
             GetLastError());

   SetupDiDestroyDeviceInfoList(hDeviceInfo);
   free(pDeviceDetail);
   exit(-1);
}

// Save the name
lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
free(pDeviceDetail);
SetupDiDestroyDeviceInfoList(hDeviceInfo);

// Open driver - Create the handle to the device
hPB3Hw4 = CreateFile(deviceName,
                     GENERIC_READ    | GENERIC_WRITE,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     NULL,
                     NULL);

if(hPB3Hw4 == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't open %s, (%d)\n", deviceName, GetLastError());
   exit(-1);
}
```

IO Controls

The driver uses IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object which controls a single board. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile(). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used. The IOCTLs defined in this driver are as follows:

IOCTL_PB3_HW4_GET_INFO

Function: Returns the Driver version, PLL ID, Switch value, Xilinx version, and Instance number.
Input: None
Output: PB3_HW4_DDINFO structure
Notes: Switch value is the configuration of the onboard dipswitch that has been selected by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDPB3Hw4.h for the definition of PB3_HW4_DDINFO.

IOCTL_PB3_HW4_SET_ACTIVE_CHANNEL

Function: Specifies the channel and offset for ReadFile or WriteFile call.
Input: Channel number and offset (PB3_HW4_MEM_ACCESS structure)
Output: None
Notes: The active channel and offset setting will remain in effect until it is overwritten.

IOCTL_PB3_HW4_PUT_DATA_WORD

Function: Writes a long word to the dual-port RAM for one channel.
Input: Channel number, memory offset, and data value to write (PB3_HW4_WRITE_WORD structure)
Output: None
Notes: This call is used to write a single long word to the data memory of one channel. All the parameters are specified in this call and the stored active channel and offset values remain unchanged.

IOCTL_PB3_HW4_GET_DATA_WORD

Function: Returns a long word value from the dual-port RAM for one channel.
Input: Channel number and offset (PB3_HW4_MEM_ACCESS structure)
Output: Data value at memory location (unsigned long integer)
Notes: This call is used to read a single long word from the data memory of one channel. All the memory parameters are specified in this call and the stored active channel and offset values remain unchanged.

IOCTL_PB3_HW4_SET_HW_CHANNEL_CONTROL
Function: Writes the configuration of a channel to its control register.
Input: Channel number and configuration parameters (PB3_HW4_HW_CNTL structure)
Output: None
Notes: See DDPB3Hw4.h for the definition of PB3_HW4_HW_CNTL.

IOCTL_PB3_HW4_GET_HW_CHANNEL_STATE
Function: Returns a channel's control configuration.
Input: Channel number (unsigned character)
Output: A channel's status values (PB3_HW4_HW_STATE structure)
Notes: See DDPB3Hw4.h for the definition of PB3_HW4_HW_STATE.

IOCTL_PB3_HW4_START_CHANNELS
Function: Starts one or more channels.
Input: Channel mask (unsigned char)
Output: None
Notes: Each bit in the input word represents one channel to be started according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc.

IOCTL_PB3_HW4_STOP_CHANNELS
Function: Stops one or more channels.
Input: Channel mask (unsigned char)
Output: None
Notes: Each bit in the input word represents one channel to be stopped according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc.

IOCTL_PB3_HW4_CHECK_CHANNELS
Function: Returns a bit-mask of the running channels.
Input: None
Output: Channel mask (unsigned char)
Notes: Each bit in the output word represents one channel that is running according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc.

IOCTL_PB3_HW4_SET_DATA
Function: Sets the data values for the 8 output bits when the data register bits are selected.
Input: Data value mask (unsigned char)
Output: None
Notes: The mux and direction bits must be in the proper state for these values to be driven onto the IO lines instead of the channel outputs.

IOCTL_PB3_HW4_GET_DATA
Function: Returns the data values for the 8 output register bits.
Input: None
Output: Data value mask (unsigned char)
Notes: This call returns the values written in the previous call.

IOCTL_PB3_HW4_SET_DIR
Function: Sets the direction of the 8 output bits when the data register bits are selected.
Input: Direction value mask (unsigned char)
Output: None
Notes: These direction controls are only valid when the corresponding mux bit value is zero. When the mux value is one, the corresponding channel state-machine controls the direction and value of the IO line.

IOCTL_PB3_HW4_GET_DIR
Function: Returns the direction values for the 8 output register bits.
Input: None
Output: Direction value mask (unsigned char)
Notes: This call returns the values written in the previous call.

IOCTL_PB3_HW4_SET_MUX
Function: Sets the state of the IO mux for the 8 IO lines.
Input: Mux value mask (unsigned char)
Output: None
Notes: When a bit is set to one the corresponding channel state-machine controls that IO line. When a bit is set to zero the state of the IO line depends on the respective direction and data bit.

IOCTL_PB3_HW4_GET_MUX
Function: Returns the state of the IO mux for the 8 IO lines.
Input: None
Output: Mux value mask (unsigned char)
Notes: This call returns the values written in the previous call.

IOCTL_PB3_HW4_READ_DATA
Function: Returns the current values of the 8 IO lines.
Input: None
Output: Data value mask (unsigned char)
Notes: This call returns the real-time value of the IO lines regardless of who is driving them.

IOCTL_PB3_HW4_GET_INT_STATUS
Function: Returns the interrupt status bit mask and clears the latched bits.
Input: None
Output: Interrupt status channel mask (unsigned char)
Notes: This command returns the mask of the channels that have an interrupt pending. Channel bits that are read as true are then cleared by writing only those bits back to the interrupt status register. This prevents missing interrupts that occur between the read and the write of the register.

IOCTL_PB3_HW4_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.

IOCTL_PB3_HW4_ENABLE_INTERRUPT

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

IOCTL_PB3_HW4_DISABLE_INTERRUPT

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

IOCTL_PB3_HW4_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 master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_PB3_HW4_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (unsigned char)
Notes: Returns the interrupt status that was read in the interrupt service routine of the last interrupt caused by one of the enabled channel interrupts. The latched status bits are cleared in the driver interrupt service routine.

IOCTL_PB3_HW4_WRITE_I2O_ADDRESS

Function: Specifies the physical address that will be used to perform the I2O accesses.
Input: unsigned long integer
Output: None
Notes: When the driver initializes it allocates some non-paged pool memory and stores the physical address of that memory in the I2O address register. That memory is then used to test the functioning of the I2O interface. This call overwrites the value in the I2O address register. It is the user's responsibility to ensure that the value written is a valid physical address for the desired location.

IOCTL_PB3_HW4_SET_I2O_CONTROL

Function: Enables the I2O interface and or clears the stored interrupt status.
Input: None
Output: PB3_HW4_I2O_CNTL structure
Notes: See DDPB3Hw4.h for the definition of PB3_HW4_I2O_CNTL.

IOCTL_PB3_HW4_I2O_TEST_READ

Function: Returns the value that was written to the stored I2O address.
Input: none
Output: I2O status value (unsigned long integer)
Notes: This call reads the external memory location that the I2O status word was written to and returns that value. This call is used to test the proper functioning of the I2O interface.

IOCTL_PB3_HW4_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: PB3_HW4_PLL_DATA structure
Output: None
Notes: The PB3_HW4_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the data to write.

IOCTL_PB3_HW4_READ_PLL_DATA
Function: Returns the contents of the PLL's internal registers
Input: None
Output: PB3_HW4_PLL_DATA structure
Notes: The register data is output in the PB3_HW4_PLL_DATA structure in an array of 40 bytes.

IOCTL_PB3_HW4_SEL_REF_CLK
Function: Selects the reference clock for the channel state-machines
Input: PLL select (BOOLEAN)
Output: None
Notes: If the input value is TRUE, the PLL clock is selected. If the input value is FALSE then the oscillator is selected.

Write

PMC-BiSerial-III-TRANS-HW4 RAM data is written to the device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the 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

PMC-BiSerial-III-TRANS-HW4 RAM data is read from the device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the 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 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, Suite 3
Santa Cruz, CA 95060
831-457-8891 Fax: 831-457-4793
support@dyneng.com