DYNAMIC ENGINEERING
435 Park Dr., Ben Lomond, Calif. 95005
831-336-8891 Fax 831-336-3840
http://www.dyneng.com
sales@dyneng.com
Est. 1988

PmcB2B

Driver Documentation

Win32 Driver Model

Revision A
Corresponding Hardware: Revision B/C
10-2004-0802/0803
Corresponding Firmware: Revision B

PmcB2B
WDM Device Driver for the
PMC-SpaceWire-BS2BI
PMC based Serial Interface

Dynamic Engineering
435 Park Drive
Ben Lomond, CA 95005
831-336-8891
831-336-3840 FAX

Manual Revision A. Revised October 3, 2006

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

The PmcB2B driver is a Win32 driver model (WDM) device driver for the PMC-SpaceWire-BS2BI from Dynamic Engineering. The PMC-SpaceWire-BS2BI is based on the PMC-SpaceWire board with some of the parts not installed. It has a Spartan3-1000 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for one serial channel in and one serial channel out. Each channel uses four LVDS signals: clock, enable, low-word serial data and high-word serial data. Each of the data lines handles 16 bits of the 32-bit parallel data word. There are two 4k x 32-bit data FIFOs, one for the transmit data and one for the receive data.

When the PMC-SpaceWire-BS2BI is recognized by the PCI bus configuration utility it will start the PmcB2B 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-SpaceWire-BS2BI user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PmcB2B.sys, PmcB2B.inf, DDPmcB2B.h, PmcB2BGUID.h, PmcB2BTest.exe, and PmcB2BTest source files.

Windows 2000 Installation

Copy PmcB2B.inf and PmcB2B.sys to a floppy disk, or CD if preferred.

With the PMC-SpaceWire-BS2BI 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 PmcB2B.inf file.
- Select *Next*.
- Select *Finish* to close the *Found New Hardware Wizard*.

Windows XP Installation

Copy PmcB2B.inf and PmcB2B.sys to a floppy disk, or CD if preferred.

With the PMC-SpaceWire-BS2BI 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 desired 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*.

The DDPmcB2B.h file is a C header file that defines the Application Program Interface (API) to the driver and contains the relevant bit defines for the PmcBS2Bi registers. The PmcB2BGUID.h file is a C header file that defines the device interface identifier for the PmcB2B driver. These files are required at compile time by any application that wishes to interface with the PmcB2B driver. They are not needed for driver installation.

The PmcB2BTest.exe file is a sample Win32 console application that makes calls into the PmcB2B driver to test each driver call without actually writing any application code. It is not required during the driver installation.

Open a command prompt console window and type ***PmcB2BTest -d0 -?*** to display a list of commands (the PmcB2BTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***PmcB2BTest -dn -im*** where ***n*** and ***m*** are the device number and driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

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 PmcB2BGUID.h.

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

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Handle to the device object
HANDLE                           hPmcB2B = INVALID_HANDLE_VALUE;
// Return status from command
LONG                             status;
// Handle to device interface information structure
HDEVINFO                         hDeviceInfo;
// Actual symbolic link name to use in the createfile
CHAR                             deviceName[MAX_DEVICE_NAME];
// Size of buffer required 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_PMC_B2B,
                          NULL,
                          NULL,
                          DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

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

interfaceData.cbSize = sizeof(interfaceData);

// Find the interface for device devNum
if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                NULL,
                        (LPGUID)&GUID_DEVINTERFACE_PMC_B2B,
                                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);
   }
}

// 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 and create the handle to the device
hPmcB2B = CreateFile(deviceName,
                     GENERIC_READ    | GENERIC_WRITE,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     NULL,
                     NULL);

if(hPmcB2B == 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_PMC_B2B_GET_INFO

Function: Returns the Driver version, Xilinx revision, Switch value and Instance number.
Input: None
Output: PMC_B2B_DRIVER_DEVICE_INFO 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). See DDPmcB2B.h for the definition of PMC_B2B_DRIVER_DEVICE_INFO.

IOCTL_PMC_B2B_SET_BASE_CONFIG

Function: Writes to the base configuration register on the PMC-SpaceWire-BS2BI.
Input: Value of control register (unsigned long integer)
Output: None
Notes: Only the bits in the BASE_CONFIG_MASK are controlled by this command. See the bit definitions in DDPmcB2B.h for information on determining this value.

IOCTL_PMC_B2B_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: Value of control register (unsigned long integer)
Notes: The return value includes the bits in BASE_CONFIG_MASK, the TX and RX enables, the DMA enables and BASE_MASTER_INT_EN.

IOCTL_PMC_B2B_GET_STATUS

Function: Returns the status register value and clears the latched status bits.
Input: None
Output: Value of status register (unsigned long integer)
Notes: Returns FIFO, IO and interrupt status. If any of the latched bits are set, this call will explicitly clear only those bits. See the bit definitions in DDPmcB2B.h for information on interpreting this value.

IOCTL_PMC_B2B_START_TX

Function: Enables the transmit state-machine to start sending data.
Input: None
Output: None
Notes: If the transmit FIFO already has data in it, this command will start the serial data transmission. If the FIFO is empty, the transmit state-machine will wait for data to be written to the FIFO. As soon as the first data-word is written the transmission will begin. If the BASE_TX_CLR_DISABLE bit in the base control register is not set, the transmit enable will automatically clear when the FIFO data is exhausted. If this bit is set, the transmission will pause, waiting for more data to be written into the FIFO.

IOCTL_PMC_B2B_STOP_TX

Function: Disables the transmit state-machine.
Input: None
Output: None
Notes: Use this call to disable the serial data transmission.

IOCTL_PMC_B2B_START_RX

Function: Enables the receive state-machine to start receiving data.
Input: None
Output: None
Notes: When the receiver is enabled, two serial data bits are received for each low to high clock transition as long as the enable signal is high. When 32 bits have been received the data-word is written to the receive FIFO and the process continues until the receiver is disabled.

IOCTL_PMC_B2B_STOP_RX

Function: Disables the receive state-machine.
Input: None
Output: None
Notes: Use this call when data reception is no longer desired.

IOCTL_PMC_B2B_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full FIFO levels.
Input: PMC_B2B_FIFO_LEVELS structure
Output: None
Notes: The FIFO levels are used to determine at what data count the TX almost empty and RX almost full status bits are asserted. See DDPB30seh.h for the definition of PMC_B2B_FIFO_LEVELS.

IOCTL_PMC_B2B_GET_FIFO_LEVELS

Function: Returns the transmitter almost empty and receiver almost full levels.
Input: None
Output: PMC_B2B_FIFO_LEVELS structure
Notes: Returns the current values for the transmit almost empty and receive almost full FIFO levels. See DDPmcB2B.h for the definition of PMC_B2B_FIFO_LEVELS.

IOCTL_PMC_B2B_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: PMC_B2B_FIFO_COUNTS structure
Notes: There is a four-deep pipeline on the output of the receive FIFO that is required for DMA processing. This means that if the receive FIFO count is not zero, there are actually four more 32-bit words available than are indicated. If the count is zero, there may be zero to four words in the pipeline. The STATUS_RX_VALID status bit read with the IOCTL_PMC_B2B_GET_STATUS call will be a one if there is valid data in the pipeline. Similarly, when the transmitter is enabled, the first word written to the transmit FIFO will be read to prepare for transmission. This reduces the transmit FIFO count by one. See DDPmcB2B.h for the definition of PMC_B2B_FIFO_COUNTS.

IOCTL_PMC_B2B_RESET_FIFOS

Function: Resets both FIFOs
Input: None
Output: None
Notes: Resets the TX and RX FIFOs.

IOCTL_PMC_B2B_WRITE_FIFO

Function: Writes a data word to the TX FIFO.
Input: FIFO word (unsigned long integer)
Output: None
Notes: This call and the following call are used to make single-word accesses to the FIFOs.

IOCTL_PMC_B2B_READ_FIFO

Function: Returns a data word from the RX FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes:

IOCTL_PMC_B2B_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 the event to be signaled.

IOCTL_PMC_B2B_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_PMC_B2B_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_PMC_B2B_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_PMC_B2B_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 interrupt status that was read in the interrupt service routine of the last interrupt caused by one of the enabled interrupts.

Write

PMC-SpaceWire-BS2BI DMA 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-SpaceWire-BS2BI DMA 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
435 Park Dr.
Ben Lomond, CA 95005
831-336-8891
831-336-3840 fax

support@dyneng.com