DYNAMIC ENGINEERING

150 DuBois, Suite C

Santa Cruz, CA 95060
(831) 457-8891
https://www.dyneng.com
sales@dyneng.com
Est. 1988

PMC-Biserial-VI-OSEH

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.19

Revision 01p2
Corresponding Hardware: Revision 03+
10-2015-0603

PMC-Biserial-VI-OSEH
WDF Device Drivers for the
PMC-BiSerial-VI-OSEH

Dynamic Engineering

150 DuBois, Suite C

Santa Cruz, CA 95060

(831) 457-8891

Table of Contents

Introduction

The PmcBis6Oseh driver was developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

PMC-BiSerial-VI-OSEH has a Spartan6 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 three RS-485 signals: clock, serial data and reference clock. There is a programmable PLL with two clock outputs. One drives the internal clock reference for the transmit state machine, the other is output on IO line 18 to be connected to IO line 2 to simulate the external clock reference. There are two 128 x 32-bit external data FIFOs, one each for the transmit and receive data. Each external FIFO is bracketed by two 4k x 32-bit internal data FIFOs for a total of 132k 32-bit words for transmit and receive functions.

UserApp is a stand-alone code set with a simple and powerful menu plus a series of tests that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering. The test software can be ported to your application to provide a running start. The regtest's are simple and will quickly demonstrate the end-to-end operation of your application making calls to the driver and interacting with the hardware.

The menu allows the user to add tests, to run sequences of tests, to run until a failure occurs and stop or to continue, to program a set number of loops to execute and more. The user can add tests to the provided test suite to try out application ideas before committing to your system configuration. In many cases the test configuration will allow faster debugging in a more controlled environment before integrating with the rest of the system.

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

Driver Installation

There are several files provided in each driver package. These files include PmcBis6OsehPublic.h, PmcBis6Oseh.inf, pmcbis6oseh.cat, and PmcBis6Oseh.sys.

PmcBis6OsehPublic.h is the C header file that defines the Application Program Interface (API) for the PmcBis6Oseh driver. This file is required at compile time by any application that wishes to interface with the drivers, but is not needed for driver installation.

Windows 10 Installation

Copy PmcBis6Oseh.inf, pmcbis6oseh.cat, and PmcBis6Oseh.sys (Win10 version) to a CD or USB memory device as preferred.

With the PMC-BiSerial-VI-OSEH hardware installed, power-on the PCI host computer.
• Open the ***Device Manager*** from the control panel.
• Under ***Other devices*** there should be an ***Other PCI Bridge Device****.
• Right-click on the ***Other PCI Bridge Device*** and select ***Update Driver Software***.
• Insert the disk or memory device prepared above in the desired drive.
• Select ***Browse my computer for driver software***.
• Select ***Let me pick from a list of device drivers on my computer***.
• Select ***Next***.
• Select ***Have Disk*** and enter the path to the device prepared above.
• Select ***Next***.
• Select ***Close*** to close the update window.
The system should now display the PmcBis6Oseh PCI adapter in the Device Manager.

*** If the *Other PCI Bridge Device*** is not displayed, click on the ***Scan for hardware changes*** icon on the tool-bar.

Driver Startup

Once the driver has been installed it will start automatically when the system recognizes the hardware.

A handle can be opened to a specific board by using the CreateFile() function call and passing in the device name obtained from the system.

The interface to the device is identified using globally unique identifiers (GUID), which are defined in PmcBis6OsehPublic.h. See main.c in the PmcBis6OsehUserApp project for an example of how to acquire a handle to the device.

The main file provided is designed to work with our test menu and includes user interaction steps to allow the user to select which board is being tested in a multiple board environment. The integrator can hardcode for single board systems or use an automatic loop to operate in multiple board systems without using user interaction. For multiple user systems it is suggested that the board number is associated with a switch setting so the calls can be associated with a particular board from a physical point of view.

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl(), and passing in the handle to the device opened with CreateFile() (see above). IOCTLs generally have input parameters, output parameters, or both. Often a custom structure is used.

```
BOOL DeviceIoControl(
  HANDLE       hDevice,          // Handle opened with CreateFile()
  DWORD        dwIoControlCode,  // Control code defined in API header
file
  LPVOID       lpInBuffer,       // Pointer to input parameter
  DWORD        nInBufferSize,    // Size of input parameter
  LPVOID       lpOutBuffer,      // Pointer to output parameter
  DWORD        nOutBufferSize,   // Size of output parameter
  LPDWORD      lpBytesReturned,  // Pointer to return length parameter
  LPOVERLAPPED lpOverlapped,     // Optional pointer to overlapped
structure
);                               //   used for asynchronous I/O
```

The IOCTLs defined for the PMC-BISERIAL-VI-OSEH driver are described below:

IOCTL_PMC_BIS6_OSEH_GET_INFO

Function: Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, and device instance number.
Input: None
Output: PMC_BIS6_OSEH_DRIVER_DEVICE_INFO structure
Notes: The switch value is the configuration of the 8-bit onboard dipswitch that has been selected by the user (see the board silk screen for bit position and polarity). Instance number is the zero-based device number. See the definition of OSEH_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _PMC_BIS6_OSEH_DRIVER_DEVICE_INFO {
   UCHAR     DriverVersion;
   UCHAR     XilinxVerMajor;
   UCHAR     XilinxVerMinor;
   UCHAR     SwitchValue;
   ULONG     InstanceNumber;
   UCHAR     PllDeviceId;
} PMC_BIS6_OSEH_DRIVER_DEVICE_INFO, *PPMC_BIS6_OSEH_DRIVER_DEVICE_INFO;
```

IOCTL_PMC_BIS6_OSEH_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input: PMC_BIS6_OSEH_PLL_DATA structure
Output: None
Notes: The PMC_BIS6_OSEH_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write. See below for the definition of PMC_BIS6_OSEH_PLL_DATA.

```
 // Structures for IOCTLs
#define PLL_MESSAGE1_SIZE        16
#define PLL_MESSAGE2_SIZE        24
#define PLL_MESSAGE_SIZE        (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _PMC_BIS6_OSEH_PLL_DATA {
   UCHAR     Data[PLL_MESSAGE_SIZE];
} PMC_BIS6_OSEH_PLL_DATA, *PPMC_BIS6_OSEH_PLL_DATA;
```

IOCTL_PMC_BIS6_OSEH_READ_PLL_DATA

Function: Reads and returns the contents of the internal registers of the PLL.
Input: None
Output: PMC_BIS6_OSEH_PLL_DATA structure
Notes: The PLL register data is returned in the PMC_BIS6_OSEH_PLL_DATA structure in an array of 40 bytes. See definition of PMC_BIS6_OSEH_PLL_DATA above.

IOCTL_PMC_BIS6_OSEH_SET_BASE_CONFIG

Function: Writes the base configuration register on the PMC-BiSerialVI-Oseh.
Input: PMC_BIS6_OSEH_SET_CONFIG structure
Output: None
Notes: The Base Configuration register data is set with the PMC_BIS6_OSEH_SET_CONFIG structure. See the bit definitions of PMC_BIS6_OSEH_SET_CONFIG below.

```
typedef struct _PMC_BIS6_OSEH_BASE_SET_CONFIG {
       BOOLEAN    TxIntEn;
       BOOLEAN    RxIntEn;
       BOOLEAN    TxAmtIntEn;
       BOOLEAN    RxAflIntEn;
       BOOLEAN    ExtClockSel;
       BOOLEAN    FifoBypass;
       BOOLEAN    TxClrDisable;
} PMC_BIS6_OSEH_BASE_SET_CONFIG, *PPMC_BIS6_OSEH_BASE_SET_CONFIG;
```

IOCTL_PMC_BIS6_OSEH_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: PMC_BIS6_OSEH_GET_CONFIG structure
Notes: The Base Configuration register data is returned in the PMC_BIS6_OSEH_GET_CONFIG structure. See the bit definitions of PMC_BIS6_OSEH_GET_CONFIG below.

```
typedef struct _PMC_BIS6_OSEH_BASE_GET_CONFIG {
       BOOLEAN    TxIntEn;
       BOOLEAN    RxIntEn;
       BOOLEAN    TxAmtIntEn;
       BOOLEAN    RxAflIntEn;
       BOOLEAN    ExtClockSel;
       BOOLEAN    FifoBypass;
       BOOLEAN    TxClrDisable;
       BOOLEAN    WrDmaIntEn;
       BOOLEAN    RdDmaIntEn;
       BOOLEAN    TxEnable;
       BOOLEAN    RxEnable;
       BOOLEAN    MIntEn;
} PMC_BIS6_OSEH_BASE_GET_CONFIG, *PPMC_BIS6_OSEH_BASE_GET_CONFIG;
```

IOCTL_PMC_BIS6_OSEH_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 when thee status is read, this call will explicitly clear only those bits. See the bit definitions below for information on interpreting this value.

```
#define STATUS_TX_FF_MT          0x00000001
#define STATUS_TX_FF_AE          0x00000002
#define STATUS_TX_FF_FL          0x00000004
#define STATUS_RX_FF_MT          0x00000010
#define STATUS_RX_FF_AF          0x00000020
#define STATUS_RX_FF_FL          0x00000040
#define STATUS_RX_VALID          0x00000080
#define STATUS_TX_INT            0x00000100
#define STATUS_RX_INT            0x00000200
#define STATUS_RX_OVFL           0x00000400
#define STATUS_LOC_INT           0x00000800
#define STATUS_WR_DMA_ERR        0x00001000
#define STATUS_RD_DMA_ERR        0x00002000
#define STATUS_WR_DMA_INT        0x00004000
#define STATUS_RD_DMA_INT        0x00008000
#define STATUS_TX_AE_LAT         0x00010000
#define STATUS_RX_AF_LAT         0x00020000
#define STATUS_INT_ACTIVE        0x80000000

#define STATUS_FIFO_MASK    (STATUS_TX_FF_MT | STATUS_TX_FF_AE | STATUS_TX_FF_FL |\
                             STATUS_RX_FF_MT | STATUS_RX_FF_AF | STATUS_RX_FF_FL |\
                             STATUS_RX_VALID)

#define STATUS_LATCH_MASK (STATUS_TX_INT | STATUS_WR_DMA_ERR | STATUS_TX_AE_LAT |\
                           STATUS_RX_INT | STATUS_RD_DMA_ERR | STATUS_RX_AF_LAT |\
                           STATUS_RX_OVFL)
```

IOCTL_PMC_BIS6_OSEH_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_BIS6_OSEH_STOP_TX

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

IOCTL_PMC_BIS6_OSEH_START_RX

Function: Enables the receive state-machine to start receiving data.
Input: None
Output: None
Notes: When the receiver is enabled, a serial data bit is received for each low to high clock transition. 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_BIS6_OSEH_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_BIS6_OSEH_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full FIFO levels.
Input: PMC_BIS6_OSEH_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. The counts are compared to the word counts of the first internal transmit FIFO and the last internal receive FIFO. Each of these FIFOs can contain up to 4095 long words, so the fields of the structure should be between 0x000 and 0xFFF. See the definition of PMC_BIS6_OSEH_FIFO_LEVELS below.

```
typedef struct _PMC_BIS6_OSEH_FIFO_LEVELS {
   USHORT   TxAlmostEmpty;
   USHORT   RxAlmostFull;
} PMC_BIS6_OSEH_FIFO_LEVELS, *PPMC_BIS6_OSEH_FIFO_LEVELS;

#define FIFO_AFL_DEF_CNT         0x00000E00  // Rx almost full level  -> 7/8 full
#define FIFO_AMT_DEF_CNT         0x00000200  // Tx almost empty level -> 1/8 full
```

IOCTL_PMC_BIS6_OSEH_GET_FIFO_LEVELS

Function: Returns the transmitter almost empty and receiver almost full levels.
Input: None
Output: PMC_BIS6_OSEH_FIFO_LEVELS structure
Notes: Returns the current values for the transmit almost empty and receive almost full FIFO levels. See the definition of PMC_BIS6_OSEH_FIFO_LEVELS above.

IOCTL_PMC_BIS6_OSEH_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: PMC_BIS6_OSEH_FIFO_COUNTS structure
Notes: There are two 4k internal FIFOs and one 128k external FIFO in each of the receiver and transmitter data paths. In addition there is a four-deep pipeline at the output of the receive FIFO chain that is required for DMA processing and a single data latch at the output of the transmit FIFO chain. Accounting for inter-FIFO latches, this means that the total transmit FIFO data is a maximum of 139,264 (0x22000) 32-bit words and the total receive FIFO data is 139,267 (0x22003) 32-bit words. The PMC_BIS6_OSEH_FIFO_COUNTS structure contains four fields. TxFF0Count and RxFF0count are the word-counts of the internal FIFOs used to determine the almost empty and almost full status; TxTotalCount and RxTotalCount are the combined counts of the entire data paths. See the definition of PMC_BIS6_OSEH_FIFO_COUNTS below.

```
typedef struct _PMC_BIS6_OSEH_FIFO_COUNTS {
   USHORT   TxFF0Count;
   ULONG    TxTotalCount;
   USHORT   RxFF0Count;
   ULONG    RxTotalCount;
} PMC_BIS6_OSEH_FIFO_COUNTS, *PPMC_BIS6_OSEH_FIFO_COUNTS;
```

IOCTL_PMC_BIS6_OSEH_RESET_FIFOS

Function: Resets all transmit and receive FIFOs.
Input: None
Output: None
Notes: Resets the TX and RX FIFO chains.

IOCTL_PMC_BIS6_OSEH_WRITE_FIFO

Function: Writes a single 32-bit 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_BIS6_OSEH_READ_FIFO

Function: Reads and returns a single 32-bit data word from the RX FIFO.
Input: None
Output: FIFO word (unsigned long integer)
Notes:

IOCTL_PMC_BIS6_OSEH_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The user 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 by the driver. The user-defined 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 unless they are explicitly enabled in the enable interrupts call.

IOCTL_PMC_BIS6_OSEH_ENABLE_INTERRUPTS

Function: Enables the DMA and/or master interrupts.
Input: PMC_BIS6_OSEH_INT_SELECT structure
Output: None
Notes: PMC_BIS6_OSEH_INT_SELECT structure has three BOOLEAN members. When WrDmaDoneInt is true, an event that has been registered with the previous call, will be signaled when a write DMA completes. Similarly, when RdDmaDoneInt is true, the event will be signaled upon the completion of a read DMA. This behavior will persist until explicitly disabled with the IOCTL_PMC_BIS6_OSEH_DISABLE_INTERRUPTS call. MasterInt enables all the other interrupts (TX, RX, FIFO levels etc.). The master interrupt is cleared in the interrupt service routine and must be re-enabled using this call after an interrupt (other than a DMA interrupt) has been serviced. See the definition of PMC_BIS6_OSEH_INT_SELECT below.

```
typedef struct _PMC_BIS6_OSEH_INT_SELECT {
   BOOLEAN  MasterInt;
   BOOLEAN  WrDmaDoneInt;
   BOOLEAN  RdDmaDoneInt;
} PMC_BIS6_OSEH_INT_SELECT, *PPMC_BIS6_OSEH_INT_SELECT;
```

IOCTL_PMC_BIS6_OSEH_DISABLE_INTERRUPTS

Function: Disables the DMA and/or master interrupt.
Input: PMC_BIS6_OSEH_INT_SELECT structure
Output: None
Notes: This call is used when DMA or user interrupt processing is no longer desired. See the definition of PMC_BIS6_OSEH_INT_SELECT above.

IOCTL_PMC_BIS6_OSEH_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_BIS6_OSEH_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 interrupt conditions.

Write

PMC-BiSerialVI-OSEH 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 an optional pointer to an Overlapped structure for performing asynchronous I/O.

It should be noted that asynchronous IO has not been tested. The size of buffer in bytes should fall on a long word boundary. The total number of writes should not exceed the number that fit in the FIFO. Writing more than will fit into the FIFO will result in data being dropped [overflow]. Fit means locations remaining in the FIFO at the time of the write command.

Read

PMC-BiSerialVI-OSEH 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 an optional pointer to an Overlapped structure for performing asynchronous I/O.

It should be noted that asynchronous IO has not been tested. The size of buffer in bytes should fall on a long word boundary. The total number of reads should not exceed the number of data in the FIFO. Reading more than stored will result in duplicated data [underflow].

Warranty and Repair

Please refer to the warranty page on our website for the current warranty offered and options.
http://www.dyneng.com/warranty.html

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 or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

Support

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois Street, Suite C
Santa Cruz, CA 95060
831-457-8891
support@dyneng.com