DYNAMIC ENGINEERING

150 DuBois, Suite C

Santa Cruz, CA 95060
(831) 457-8891 **Fax** (831) 457-4793
http://www.dyneng.com
sales@dyneng.com
Est. 1988

Pmc Bis3 Hw2

Software Manual

Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Revision A1
10-2006-0803

PmcBis3Hw2
WDF Device Drivers for the
PMC Biserial 3 Hw2

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

Manual Revision A. Revised September 12, 2018.

Table of Contents

Introduction

The Pmc-BiSerial-III-HW2 driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF). It was developed using 64 bit Windows operating system with an Intel Core i7 Processor.

The PMC-BiSerial-III-HW2 board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, Dual-Port RAM and protocol control and status for up to 32 channels. There is also a programmable PLL with two clock outputs that are used as the clock reference for the SDLC and asynchronous interfaces. Each channel has one or more 2k-byte dual-port RAM for data transmission and reception.

The 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 register tests 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 the PMC-BiSerial-III-HW2 is recognized by the PCI bus configuration utility it will start the PmcBis3Hw2 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-III-Hw2 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include PmcBis3Hw2Public.h, PmcBis3Hw2.inf, pmcbis3hw2.cat, PmcBis3Hw2.sys, and WdfCoInstaller01009.dll.

PmcBis3Hw2Public.h is the C header file that define the Application Program Interface (API) for the PmcBis3Hw2 driver. These files are required at compile time by any application that wishes to interface with the drivers, but is not needed for driver installation.

Windows 7 Installation

Copy PmcBis3Hw2.inf, pmcbis3hw2.cat, PmcBis3Hw2.sys, and WdfCoInstaller01009.dll (Win7 version) to a CD or USB memory device as preferred.

With the PMC BIS3 HW2 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.
- Follow the same steps to install the channel drivers.

The system should now display the PmcBis3Hw2 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 PmcBis3Hw2Public.h. See main.c in the PmcBis3Hw2UserApp 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 PmcBis3Hw2 driver are described below:

IOCTL_PMC_BIS3_HW2_BASE_GET_INFO

Function: Returns the current driver version and instance number.
Input: none
Output: PMC_BIS3_HW2_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). 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 the definition of PMC_BIS3_HW2_DRIVER_DEVICE_INFO below. Refer to the PrintInfo function found in the PrintInfo.c file in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_DRIVER_DEVICE_INFO {
   UCHAR    DriverVersion;
   UCHAR    SwitchValue;
   UCHAR    PllDeviceId;
   USHORT   DesignId;
   USHORT   DesignRev;
   ULONG    InstanceNumber;
} PMC_BIS3_HW2_DRIVER_DEVICE_INFO, *PPMC_BIS3_HW2_DRIVER_DEVICE_INFO;
```

IOCTL_PMC_BIS3_HW2_SET_CHANNEL_MODE

Function: Specifies the mode of operation for the eight groups of four channels
Input: PMC_BIS3_HW2_MODE structure
Output: none
Notes: There are three modes of operation for the channel groups: SDLC, ASYNC or HW mode. Currently the first two channel blocks can only be set to HW mode and the remaining six blocks can be set to either SDLC or ASYNC mode. See the definition of PMC_BIS3_HW2_MODE_TYPE and PMC_BIS3_HW2_MODE below. Refer to the async_test or sdlc_test function found in the UserApp for an example of use.

```
typedef enum _PMC_BIS3_HW2_MODE_TYPE {
   PMC_BIS3_HW2_SDLC,
   PMC_BIS3_HW2_ASYNC,
   PMC_BIS3_HW2_HW
} PMC_BIS3_HW2_MODE_TYPE, *PPMC_BIS3_HW2_MODE_TYPE;

typedef struct _PMC_BIS3_HW2_MODE {
   PMC_BIS3_HW2_MODE_TYPE   ChanBlockModes[PMC_BIS3_HW2_NUM_CHANBLOCKS];
} PMC_BIS3_HW2_MODE, *PPMC_BIS3_HW2_MODE;
```

IOCTL_PMC_BIS3_HW2_GET_CHANNEL_MODE

Function: Returns the mode of operation for the eight groups of four channels.
Input: none
Output: PMC_BIS3_HW2_MODE structure
Notes: This call returns the values written in the previous call. See the definition of PMC_BIS3_HW2_MODE_TYPE and PMC_BIS3_HW2_MODE above.

IOCTL_PMC_BIS3_HW2_SET_ACTIVE_CHANNEL

Function: Specifies the channel and offset for ReadFile or WriteFile call
Input: Channel number and offset (PMC_BIS3_HW2_MEM_ACCESS structure)
Output: none
Notes: The active channel and offset setting will remain in effect until it is overwritten. See the definition of PMC_BIS3_HW2_MEM_ACCESS below. Refer to the hsloop_tst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_MEM_ACCESS {
   UCHAR       Channel;
   USHORT      Offset;
} PMC_BIS3_HW2_MEM_ACCESS, *PPMC_BIS3_HW2_MEM_ACCESS;
```

IOCTL_PMC_BIS3_HW2_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 (PMC_BIS3_HW2_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. See the definition of PMC_BIS3_HW2_WRITE_WORD below. Refer to the hsloop_tst function found in the UserApp for an example of use

```
typedef struct _PMC_BIS3_HW2_WRITE_WORD {
   UCHAR       Channel;
   USHORT      Offset;
   ULONG       Data;
} PMC_BIS3_HW2_WRITE_WORD, *PPMC_BIS3_HW2_WRITE_WORD;
```

IOCTL_PMC_BIS3_HW2_GET_DATA_WORD

Function: Returns a long word value from the dual-port RAM for one channel.
Input: Channel number and offset (PMC_BIS3_HW2_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. See the definition of PMC_BIS3_HW2_MEM_ACCESS with the SET_ACTIVE_CHANNEL call above. Refer to the hsloop_tst function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_SET_HW_CHANNEL_CONTROL

Function: Writes the configuration of an HW1 channel to its control register.
Input: Channel number and configuration parameters (PMC_BIS3_HW2_HW_CNTL structure)
Output: None
Notes: See the definition of PMC_BIS3_HW2_HW_CNTL structure below. Refer to the hsloop_tst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_HW_CNTL {
   UCHAR       Channel;
   USHORT      EndOfMessage;
   BOOLEAN     Transmit;
   BOOLEAN     HighSpeed;
   BOOLEAN     BiDirectional;
   BOOLEAN     InsertIdles;
   BOOLEAN     ClearEnable;
   BOOLEAN     IntEnable;
} PMC_BIS3_HW2_HW_CNTL, *PPMC_BIS3_HW2_HW_CNTL;
```

IOCTL_PMC_BIS3_HW2_GET_HW_CHANNEL_CONTROL

Function: Returns a channel's control configuration.
Input: Channel number (unsigned character)
Output: A channel's status values (PMC_BIS3_HW2_HW_STATE structure)
Notes: See the definition of PMC_BIS3_HW2_HW_STATE below. Refer to the hsloop_tst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_HW_STATE {
   USHORT      EndAddress;
   USHORT      EndOfMessage;
   BOOLEAN     Transmit;
   BOOLEAN     HighSpeed;
   BOOLEAN     BiDirectional;
   BOOLEAN     InsertIdles;
   BOOLEAN     ClearEnable;
   BOOLEAN     IntEnable;
   BOOLEAN     ManError;
   BOOLEAN     PostAmbleError;
   BOOLEAN     CrcError;
   BOOLEAN     Ready;
} PMC_BIS3_HW2_HW_STATE, *PPMC_BIS3_HW2_HW_STATE;
```

IOCTL_PMC_BIS3_HW2_START_CHANNELS

Function: Starts one or more channels.
Input: Channel mask (unsigned long integer)
Output: None
Notes: Each bit in the input word represents one HW channel to be started according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc. Only bits 0 – 7 are used by this IOCTL. Refer to the hsloop_tst function

IOCTL_PMC_BIS3_HW2_STOP_CHANNELS

Function: Stops one or more channels.
Input: Channel mask (unsigned long integer)
Output: None
Notes: Each bit in the input word represents one HW channel to be stopped according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc. Only bits 0 – 7 are used by this IOCTL. Refer to the hsloop_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_CHECK_CHANNELS

Function: Returns a bit-mask of the running channels.
Input: None
Output: Channel mask (unsigned long integer)
Notes: Each bit in the output word represents one HW channel that is running according to its position. Bit 0 represents channel 0, bit 1 represents channel 1, etc. Only bits 0 – 7 are used by this IOCTL.

IOCTL_PMC_BIS3_HW2_SET_SDLC_CHANNEL_CONTROL

Function: Writes the configuration of a channel to its control register.
Input: Channel number and configuration parameters (PMC_BIS3_HW2_SDLC_CNTL structure)
Output: None
Notes: See the definition of PMC_BIS3_HW2_SDLC_CNTL below. The LoadTx/RxAddress fields are used when multiple frames are sent or received. When FALSE, new starting addresses aren't loaded and read/writes will follow the previous message-frame. Refer to the sdlc_lptst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_SDLC_CNTL {
   UCHAR       Channel;
   BOOLEAN     TxEnable;
   BOOLEAN     RxEnable;
   BOOLEAN     TxClearEnable;
   BOOLEAN     TxIntEnable;
   BOOLEAN     TxFrmDnIntEn;
   BOOLEAN     RxIntEnable;
   BOOLEAN     AbortIntEnable;
   BOOLEAN     TxFlgsShrZero;
   USHORT      TxStartAddress;
   USHORT      RxStartAddress;
   USHORT      TxEndAddress;
   BOOLEAN     TxIdleFrmEnd;
   BOOLEAN     LdRxStartAdd;
   BOOLEAN     LdTxStartAdd;
   BOOLEAN     LdTxEndAdd;
   BOOLEAN     SendAbort;
   BOOLEAN     AbortClear;
   BOOLEAN     IdleClear;
} PMC_BIS3_HW2_SDLC_CNTL, *PPMC_BIS3_HW2_SDLC_CNTL;
```

IOCTL_PMC_BIS3_HW2_GET_SDLC_CHANNEL_STATE

Function: Returns a channel's status and configuration.
Input: Channel number (unsigned character)
Output: A channel's status values (PMC_BIS3_HW2_SDLC_STATE structure)
Notes: See the definition of PMC_BIS3_HW2_SDLC_STATE below. Refer to the sdlc_lptst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_SDLC_STATE {
   BOOLEAN     TxEnable;
   BOOLEAN     RxEnable;
   USHORT      RxEndAddress;
   BOOLEAN     TxSndngFrm;
   BOOLEAN     TxFrmDone;
   BOOLEAN     TxClearEnable;
   BOOLEAN     TxIntEnable;
   BOOLEAN     TxDnIntEnable;
   BOOLEAN     RxIntEnable;
   BOOLEAN     AbortIntEnable;
   BOOLEAN     TxFlgsShrZero;
   BOOLEAN     TxIdleFrmEnd;
   BOOLEAN     AbortReceived;
   BOOLEAN     IdleDetected;
} PMC_BIS3_HW2_SDLC_STATE, *PPMC_BIS3_HW2_SDLC_STATE;
```

IOCTL_PMC_BIS3_HW2_SET_ASYNC_CHANNEL_CONTROL

Function: Writes the configuration of a channel to its control register.
Input: Channel number and configuration parameters (PMC_BIS3_HW2_ASYNC_CNTL structure)
Output: None
Notes: See the definition of PMC_BIS3_HW2_ASYNC_CNTL below. Refer to the async_lptst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_ASYNC_CNTL {
   UCHAR       Channel;
   BOOLEAN     TxEnable;
   BOOLEAN     RxEnable;
   USHORT      TxStartAddress;
   USHORT      TxEndAddress;
   USHORT      RxStartAddress;
   BOOLEAN     PllBClkSel;
   BOOLEAN     TxClearEnable;
   BOOLEAN     TxIntEnable;
   BOOLEAN     RxIntEnable;
} PMC_BIS3_HW2_ASYNC_CNTL, *PPMC_BIS3_HW2_ASYNC_CNTL;
```

IOCTL_PMC_BIS3_HW2_GET_ASYNC_CHANNEL_STATE

Function: Returns a channel's status and control configuration.
Input: Channel number (unsigned character)
Output: A channel's status values (PMC_BIS3_HW2_ASYNC_STATE structure)
Notes: See the definition of PMC_BIS3_HW2_ASYNC_STATE below. Refer to the async_lptst function found in the UserApp for an example of use.

```
typedef struct {
   BOOLEAN     TxEnable;
   BOOLEAN     RxEnable;
   BOOLEAN     TxClearEnable;
   BOOLEAN     PllBClkSel;
   BOOLEAN     TxIntEnable;
   BOOLEAN     RxIntEnable;
   USHORT      RxEndAddress;
   BOOLEAN     FrameError;
} PMC_BIS3_HW2_ASYNC_STATE, *PPMC_BIS3_HW2_ASYNC_STATE;
```

IOCTL_PMC_BIS3_HW2_SET_DATA

Function: Sets the data values for the 34 output bits when the data register bits are selected.
Input: Data value mask (PMC_BIS3_HW2_IO structure)
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. See the definition of PMC_BIS3_HW2_IO below. Refer to the iolp_tst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_IO {
   ULONG       IoData;
   BOOLEAN     Bit32;
   BOOLEAN     Bit33;
} PMC_BIS3_HW2_IO, *PPMC_BIS3_HW2_IO;
```

IOCTL_PMC_BIS3_HW2_GET_DATA

Function: Returns the data values for the 34 output register bits.
Input: None
Output: Data value mask (PMC_BIS3_HW2_IO structure)
Notes: This call returns the values written in the previous call. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_SET_DIR

Function: Sets the direction of the 34 output bits when the data register bits are selected.
Input: Direction value mask (PMC_BIS3_HW2_IO structure)
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. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_GET_DIR

Function: Returns the direction values for the 34 output register bits.
Input: None
Output: Direction value mask (PMC_BIS3_HW2_IO structure)
Notes: This call returns the values written in the previous call. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_SET_TERM

Function: Sets the termination of the 34 IO lines.
Input: Termination value mask (PMC_BIS3_HW2_IO structure)
Output: None
Notes: The terminations are switched in or out based on the values written in this call. They are independent of the mux and direction bits. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_GET_TERM

Function: Returns the termination values for the 34 IO lines.
Input: None
Output: Termination value mask (PMC_BIS3_HW2_IO structure)
Notes: This call returns the values written in the previous call. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_SET_MUX

Function: Sets the state of the IO mux for the 34 IO lines.
Input: Mux value mask (PMC_BIS3_HW2_IO structure)
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. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_GET_MUX

Function: Returns the state of the IO mux for the 34 IO lines.
Input: None
Output: Mux value mask (PMC_BIS3_HW2_IO structure)
Notes: This call returns the values written in the previous call. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_READ_DATA

Function: Returns the current values of the 34 IO lines.
Input: None
Output: Data value mask (PMC_BIS3_HW2_IO structure)
Notes: This call returns the real-time value of the IO lines regardless of who is driving them. See the definition of PMC_BIS3_HW2_IO above. Refer to the iolp_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_GET_INT_STATUS

Function: Returns the interrupt status bit mask and clears the latched bits.
Input: None
Output: Interrupt status channel mask (unsigned long integer)
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 thus preventing missed interrupts that occur between the read and the write of the register. Refer to the hsloop_tst function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_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(). The returned handle is the input to this IOCTL. The driver then 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. Refer to the interrupt function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_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 an interrupt occurs to be ready for the next interrupt. Refer to the interrupt function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_DISABLE_INTERRUPT

Function: Disables the master interrupt.
Input: None
Output: None
Notes: This call is used when local interrupt processing is no longer desired. Refer to the interrupt function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_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. Refer to the interrupt function found in the UserApp for an example of use

IOCTL_PMC_BIS3_HW2_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 channel interrupts. The latched status bits are cleared in the driver interrupt service routine. Refer to the interrupt function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_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. Refer to the i20_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_SET_I2O_CONTROL

Function: Enables the I2O interface and or clears the stored interrupt status.
Input: None
Output: PMC_BIS3_HW2_I2O_CNTL structure
Notes: See the definition of PMC_BIS3_HW2_I2O_CNTL below. Refer to the i20_tst function found in the UserApp for an example of use.

```
typedef struct _PMC_BIS3_HW2_I2O_CNTL {
   BOOLEAN     Enable;
   BOOLEAN     Clear;
} PMC_BIS3_HW2_I2O_CNTL, *PPMC_BIS3_HW2_I2O_CNTL;
```

IOCTL_PMC_BIS3_HW2_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. Refer to the i20_tst function found in the UserApp for an example of use.

IOCTL_PMC_BIS3_HW2_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: PMC_BIS3_HW2_PLL_DATA structure
Output: None
Notes: The PMC_BIS3_HW2_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the data to write. See the definition of PMC_BIS3_HW2_PLL_DATA below. Refer to the PLL_if_test function found in the UserApp for an example of use.

```
#define PLL_MESSAGE1_SIZE           16
#define PLL_MESSAGE2_SIZE           24
#define PLL_MESSAGE_SIZE           (PLL_MESSAGE1_SIZE +
PLL_MESSAGE2_SIZE)

typedef struct _PMC_BIS3_HW2_PLL_DATA {
   UCHAR   Data[PLL_MESSAGE_SIZE];
} PMC_BIS3_HW2_PLL_DATA, *PPMC_BIS3_HW2_PLL_DATA;
```

IOCTL_PMC_BIS3_HW2_READ_PLL_DATA

Function: Returns the contents of the PLL's internal registers
Input: None
Output: PMC_BIS3_HW2_PLL_DATA structure
Notes: The register data is output in the PMC_BIS3_HW2_PLL_DATA structure in an array of 40 bytes. Refer to the PLL_if_test function found in the UserApp for an example of use.

Write

PMC-BiSerial-III-HW2 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-HW2 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

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
831-457-4793 Fax
support@dyneng.com