DYNAMIC ENGINEERING

150 DuBois, Suite C

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

(cc)PMC-BiSerial-VI-GPIO
Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation Ver1.19

Revision 01p1 11/10/21
PMC: 10-2015-0604/5
ccPMC: 10-2021-0401/2

BiSerial-VI-GPIO
WDF Device Drivers for
PMC-BiSerial-VI-GPIO
ccPMC-BiSerial-VI-GPIO

Dynamic Engineering

150 DuBois, Suite C

Santa Cruz, CA 95060

(831) 457-8891

Table of Contents

Introduction

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

PMC-Biserial-VI-GPIO and ccPMC-BiSerial-VI-GPIO are supported with the same driver interface. “BiSerial-VI-GPIO” features a Spartan6 Xilinx FPGA to implement the PCI interface, FIFOs, and IO processing, control and status for 32 differential IO. Each IO can be RS-485 or LVDS (build option). There is a programmable PLL with four clock outputs. PLLA or the 50 MHz reference can be used as the for the COS clock divider. Many COS frequencies are user selectable in this manner. An unusual feature is a standalone FIFO [8Kx32] with DMA in and out. The memory doesn’t “do anything” since it is not currently attached to input or output data. It is available for user purposes and to support future requirements.

UserAp 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 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. The test menu uses the Type field to know which board type it is communicating with and prints that out at the top of the menu. The .inf file also has the definitions and the system will show the type in the device manager after installation of the driver.

UserAp is delivered with multiple example tests plus 2 utilities that may prove useful in your debugging / integration. At the end of the UserAp menu is an item “Print Registers”. When executed – select the appropriate “test” number in the menu – the current contents of the registers are displayed. The structures for the Base and Base 1 registers are shown with the structure selection and current status. The remainder are shown as hex numbers. Easy way to check if GPIO is set-up the way you think it is.

The second utility is “Modify Registers”. This utility allows the user to select a register to change, shows the current contents and allows the user to change the contents. The utility will allow multiple changes to the same register, switching to a new register. For example, one can enable selected outputs, and set and change the IO definition.

When BiSerial-VI-GPIO is recognized by the PCI bus configuration utility it will start the ccPmcBiSerialViGpio 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-Parallel-TTL-GPIO or XMC-Parallel-TTL-GPIO user manual as appropriate (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include ccPmcBiSerialViGpioPublic.h, ccPmcBiSerialViGpio.inf, ccPmcBiSerialViGpio.cat, and ccPmcBiSerialViGpio.sys.

ccPmcBiSerialViGpioPublic.h is the C header file that defines the Application Program Interface (API) for the ccPmcBiSerialViGpio 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. This file is included with the UserAp file set.

Windows 10 Installation

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

With the BiSerial-VI-GPIO hardware installed, power-on the 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.

• Select ***Browse my computer for driver software***.
• Select ***Navigate to the folder or device. If at the root select the sub folders button.***
• Select ***Next***.
• Select ***Close*** to close the update window.
The system should now display the ccPmcBiSerialViGpio adapter in the Device Manager. The type will also be shown – PMC or ccPMC

***** 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 ccPmcBiSerialViGpioPublic.h. See main.c in the ParallelTtlGpioUserAp 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 Win 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 Parallel-TTL-GPIO driver are described below:
66 currently defined. Two part IOCTLs are grouped – Upper and Lower access to 64 bit registers etc.

IOCTL_PMC_BIS6_GPIO_GET_INFO

Function: Returns the device driver version, Xilinx flash revision, PLL device ID, user switch value, Type, and device instance number.
Input: None
Output: PAR_TTL_GPIO_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. Revision Major and Revision Minor represent the current Flash revision Major.Minor. Type is set to 1 or 2 to show if PMC or XMC respectively. PLL Device ID is the I2C address discovered.

```
// Driver/Device information
typedef struct _PAR_TTL_GPIO_DRIVER_DEVICE_INFO {
  ULONG        InstanceNumber;
  UCHAR        DriverVersion;
  UCHAR        RevisionMajor;
  UCHAR        RevisionMinor;
  UCHAR        SwitchValue;
  UCHAR        TypeValue;
  UCHAR        PllDeviceId;
  BOOLEAN      BridgeConfigured;
} PAR_TTL_GPIO_DRIVER_DEVICE_INFO, *PPAR_TTL_GPIO_DRIVER_DEVICE_INFO;
```

IOCTL_PMC_BIS6_GPIO_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input: PAR_TTL_GPIO_PLL_DATA structure
Output: None
Notes: The PAR_TTL_GPIO_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 PAR_TTL_GPIO_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 _PAR_TTL_GPIO_PLL_DATA {
   UCHAR    Data[PLL_MESSAGE_SIZE];
} PAR_TTL_GPIO_PLL_DATA, *PPAR_TTL_GPIO_PLL_DATA;
```

IOCTL_PMC_BIS6_GPIO_READ_PLL_DATA

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

IOCTL_PMC_BIS6_GPIO_SET_BASE_CONFIG

Function: Writes the base configuration register on the Parallel-TTL-GPIO.
Input: PAR_TTL_GPIO_SET_CONFIG structure
Output: None
Notes: The Base Configuration register data is set with the PAR_TTL_GPIO_SET_CONFIG structure.

```
typedef struct _PAR_TTL_GPIO_BASE_SET_CONFIG {
        BOOLEAN   IoRst; // set to cause reset, return to cleared required to operate
        BOOLEAN   ForceIntEn; // set to force interrupt, clear to remove
        BOOLEAN   CosClkSel; // set to use PLL, cleared uses oscillator reference
        BOOLEAN   TestClkSel; // not set = std operation, set = push reference clock onto IO
} PAR_TTL_GPIO_BASE_SET_CONFIG, *PPAR_TTL_GPIO_BASE_SET_CONFIG;
```

IOCTL_PMC_BIS6_GPIO_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: PAR_TTL_GPIO_GET_CONFIG structure
Notes: The Base Configuration register data is returned in the PAR_TTL_GPIO_GET_CONFIG structure.

```
typedef struct _PAR_TTL_GPIO_BASE_GET_CONFIG {
        BOOLEAN   IoRst; // set to cause reset, return to cleared required to operate
        BOOLEAN   ForceIntEn; // set to force interrupt, clear to remove
        BOOLEAN   CosClkSel; // set to use PLL, cleared uses oscillator reference
        BOOLEAN   TestClkSel; // not set = std operation, set = push reference clock onto IO
} PAR_TTL_GPIO_BASE_GET_CONFIG, *PPAR_TTL_GPIO_BASE_GET_CONFIG;
```

IOCTL_PMC_BIS6_GPIO_GET_STATUS

Function: Returns the status register value
Input: None
Output: Value of status register (unsigned long integer)
Notes: Returns FIFO, IO and interrupt status. See HW manual for detail about the meaning of the bits. Please note: Public.h contains some additional bits reserved for when FIFO based IO is implemented. Those bits are set to ‘0’ currently.

IOCTL_PMC_BIS6_GPIO_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full FIFO levels.
Input: PAR_TTL_GPIO_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 transmit FIFO or receive FIFO. The FIFOs are not currently in use and have been replaced with a single FIFO for DMA demonstration. See the definition of PAR_TTL_GPIO_FIFO_LEVELS below.

```
typedef struct _PAR_TTL_GPIO_FIFO_LEVELS {
   USHORT   TxAlmostEmpty;
   USHORT   RxAlmostFull;
} PAR_TTL_GPIO_FIFO_LEVELS, *PPAR_TTL_GPIO_FIFO_LEVELS;
```

IOCTL_PMC_BIS6_GPIO_GET_FIFO_LEVELS

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

IOCTL_PMC_BIS6_GPIO_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive FIFOs.
Input: None
Output: PAR_TTL_GPIO_FIFO_COUNTS structure
Notes: Currently 8Kx32 FIFO plus a four-deep pipeline at the output of the receive FIFO chain that is required for DMA processing. The PAR_TTL_GPIO_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. Currently only the RxTotalCount is valid. Others are set to “0”.

```
typedef struct _PAR_TTL_GPIO_FIFO_COUNTS {
   USHORT   TxFF0Count;
   ULONG    TxTotalCount;
   USHORT   RxFF0Count;
   ULONG    RxTotalCount;
} PAR_TTL_GPIO_FIFO_COUNTS, *PPAR_TTL_GPIO_FIFO_COUNTS;
```

IOCTL_PMC_BIS6_GPIO_RESET_FIFOS

Function: Resets all transmit and receive FIFOs.
Input: None
Output: None
Notes: Automatically toggles the IoRst bit in the base register to reset the statemachines, FIFO pointers etc.

IOCTL_PMC_BIS6_GPIO_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_GPIO_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: Tx and Rx FIFO are currently the same FIFO.

IOCTL_PMC_BIS6_GPIO_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_GPIO_ENABLE_INTERRUPTS

Function: Enables the DMA and/or master interrupts.
Input: PAR_TTL_GPIO_INT_SELECT structure
Output: None
Notes: PAR_TTL_GPIO_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_GPIO_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 PAR_TTL_GPIO_INT_SELECT below. See also IO Interrupt Enables.

```
typedef struct _PAR_TTL_GPIO_INT_SELECT {
   BOOLEAN  MasterInt;
   BOOLEAN  WrDmaDoneInt;
   BOOLEAN  RdDmaDoneInt;
} PAR_TTL_GPIO_INT_SELECT, *PPAR_TTL_GPIO_INT_SELECT;
```

IOCTL_PMC_BIS6_GPIO_DISABLE_INTERRUPTS

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

IOCTL_PMC_BIS6_GPIO_FORCE_INTERRUPT

Function: Causes a system interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted as long as the master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_PMC_BIS6_GPIO_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: **PAR_TTL_GPIO_ISR_STAT** structure
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. Multiple ULONG members with complete status for the Interrupt Status register, Rising, Falling, and Filtered data.

```
typedef struct _PAR_TTL_GPIO_ISR_STAT {
        ULONG   InterruptStatus;
        ULONG   RisingData0;
        ULONG   RisingData1;
        ULONG   FallingData0;
        ULONG   FallingData1;
        ULONG   FilteredData0;
        ULONG   FilteredData1;
} PAR_TTL_GPIO_ISR_STAT, * PPAR_TTL_GPIO_ISR_STAT;
```

IOCTL_PMC_BIS6_GPIO_BRIDGE_RECONFIG

Function: Look for upstream TSI284 bridge and reprogram
Input: None
Output: None
Notes: Creates a work item that looks for an upstream bridge. For example if the XMC is used the bridge is on the XMC. If the PMC is used with PCIeBPMCX1 the bridge is on the carrier. Certain settings are modified to enhance DMA performance. To see if configuration was successful [BridgeConfigured] check that status. Since the work item operates in parallel allow for this call to complete. Example in the menu.

IOCTL_PMC_BIS6_GPIO_GET_INT_COUNT

Function: Read current count of interrupts from this device
Input: None
Output: **INT_COUNT** Structure
Notes:

```
typedef struct _INT_COUNT {
    ULONG     int_count;
    ULONG     force_int_count;
} INT_COUNT, *PINT_COUNT;
```

IOCTL_PMC_BIS6_GPIO_SET_MASTER_INT_CONFIG

Function: Enable or Disable Master Interrupt Enable
Input: **PAR_TTL_GPIO_MASTER_INT_CONFIG**
Output: None
Notes:

IOCTL_PMC_BIS6_GPIO_GET_MASTER_INT_CONFIG

Function: Reads and returns state of Master Interrupt Enable
Input: None
Output: **PAR_TTL_GPIO_MASTER_INT_CONFIG**
Notes:

```
typedef struct _PAR_TTL_GPIO_MASTER_INT_CONFIG {
        BOOLEAN       MasterIntEn;
} PAR_TTL_GPIO_MASTER_INT_CONFIG, * PPAR_TTL_GPIO_MASTER_INT_CONFIG;
```

//

IOCTL_PMC_BIS6_GPIO_SET_DATA_OUT0

Function: Writes a single 32-bit data-word to the Data Register
Input: ULONG
Output: None
Notes: If the IO is enabled to be updated the data will flow to the output based on the Enable Registers.

IOCTL_PMC_BIS6_GPIO_GET_DATA_OUT0

Function: Reads and returns a single 32-bit data word from the Data Register.
Input: None
Output: ULONG
Notes: This is the register read-back and will match the SET data.

IOCTL_PMC_BIS6_GPIO_SET_EN0

Function: Writes a single 32-bit data-word to the Data Enable Register
Input: ULONG
Output: None
Notes: For each bit set to ‘1’ the driver will be enabled. The corresponding bit in the DATA_OUT registers will be driven to the external IO. Bits not set are not driven.

IOCTL_PMC_BIS6_GPIO_GET_EN0

Function: Reads and returns a single 32-bit data word from the Data Enable Register.
Input: None
Output: ULONG
Notes:

IOCTL_PMC_BIS6_GPIO_SET_POLARITY0

Function: Writes a single 32-bit data-word to the Polarity Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit will be inverted. Only affects input side data, not the driven data. See the FilteredData registers.

IOCTL_PMC_BIS6_GPIO_GET_POLARITY0

Function: Reads and returns a single 32-bit data word from the Polarity Register.
Input: None
Output: ULONG
Notes:

IOCTL_PMC_BIS6_GPIO_SET_EDGE_LEVEL0

Function: Writes a single 32-bit data-word to the EdgeLevel Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the bit will be treated as edge sensitive. Only affects input side data, not the driven data. For each bit cleared, the data is treated as level sensitive.

IOCTL_PMC_BIS6_GPIO_GET_EDGE_LEVEL0

Function: Reads and returns a single 32-bit data word from the EdgeLevel Register.
Input: None
Output: ULONG
Notes:

IOCTL_PMC_BIS6_GPIO_SET_INT_EN0

Function: Writes a single 32-bit data-word to the Interrupt Enable Register
Input: ULONG
Output: None
Notes: For each bit set to ‘1’ the bit the associated interrupt will be enabled. Only affects input side data. Used for both Level and Edge defined processing. See Rising and Falling for additional options.

IOCTL_PMC_BIS6_GPIO_GET_INT_EN0

Function: Reads and returns a single 32-bit data word from the Interrupt Enable Register.
Input: None
Output: ULONG
Notes:

IOCTL_PMC_BIS6_GPIO_READ_DIRECT0

Function: Reads and returns a single 32-bit data word from the IO port.
Input: None
Output: ULONG
Notes: Direct data is synchronized but not filtered in any way. Get the state of the IO (whether defined as output or input).

IOCTL_PMC_BIS6_GPIO_READ_FILTERED0

Function: Reads and returns a single 32-bit data word from the IO port after manipulation.
Input: None
Output: ULONG
Notes: Data is synchronized and filtered. Polarity and EdgeLevel are applied.

IOCTL_PMC_BIS6_GPIO_SET_COS_RISING_STAT0

Function: Writes a single 32-bit data-word to the Rising Status Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding bit in the Rising Status Register is cleared.

IOCTL_PMC_BIS6_GPIO_GET_COS_RISING_STAT0

Function: Reads and returns a single 32-bit data word from the Rising Status Register.
Input: None
Output: ULONG
Notes: When an IO bit programmed as Edge and Rising transitions from low to high the status bit is set. If the corresponding Interrupt Enable is also set an interrupt is generated. Clear by writing back with the bit(s) set.

IOCTL_PMC_BIS6_GPIO_SET_COS_FALLING_STAT0

Function: Writes a single 32-bit data-word to the Falling Status Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding bit in the Falling Status Register is cleared.

IOCTL_PMC_BIS6_GPIO_GET_COS_FALLING_STAT0

Function: Reads and returns a single 32-bit data word from the Falling Status Register.
Input: None
Output: ULONG
Notes: When an IO bit programmed as Edge and Falling transitions from HIgh to Low the status bit is set. If the corresponding Interrupt Enable is also set an interrupt is generated. Clear by writing back with the bit(s) set.

IOCTL_PMC_BIS6_GPIO_SET_COS_RISING_EN0

Function: Writes a single 32-bit data-word to the Rising Enable Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding IO bit is enabled to be captured for rising edge transitions.

IOCTL_PMC_BIS6_GPIO_GET_COS_RISING_EN0

Function: Reads and returns a single 32-bit data word from the Rising Enable Register.
Input: None
Output: ULONG
Notes: Register read, will match current register value.

IOCTL_PMC_BIS6_GPIO_SET_COS_FALLING_EN0

Function: Writes a single 32-bit data-word to the Falling Enable Register
Input: ULONG
Output: None
Notes: For each bit set to '1' the corresponding IO bit is enabled to be captured for falling edge transitions.

IOCTL_PMC_BIS6_GPIO_GET_COS_FALLING_EN0

Function: Reads and returns a single 32-bit data word from the Falling Enable Register.
Input: None
Output: ULONG
Notes: Register read, will match current register value.

IOCTL_PMC_BIS6_GPIO_SET_HALFDIV

Function: Writes a single 32-bit data-word to the Rising Enable Register
Input: ULONG
Output: None
Notes: Write to this register to define divider to apply to COS reference clock selected. COS clock is Reference / 2N where N= 16 bits. Set upper bits to 0.

IOCTL_PMC_BIS6_GPIO_GET_HALFDIV

Function: Reads and returns a single 32-bit data word from the HalfDiv Register.
Input: None
Output: ULONG
Notes: Register read, will match current register value.

IOCTL_PMC_BIS6_GPIO_SET_TMP

Function: Writes a single 32-bit data-word to the Rising Enable Register
Input: None
Output: None
Notes: Write to this register to initiate read of temperature data

IOCTL_PMC_BIS6_GPIO_GET_TMP

Function: Read from LM75 port
Input: None
Output: ULONG
Notes: More information in HW manual and Temp.c example code.

Write

BiSerial-VI-GPIO DMA data is written to the device using the write command. Writes are executed using the Win32 function DE_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

BiSerial-VI-GPIO DMA data is read from the device using the read command. Reads are executed using the Win32 function DE_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].

Please note: Windows 10 has some bugs.   With the 20H2 revision many of the issues affecting proper DMA operation have been resolved. It is expected this revision or later Win10 is in use.

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