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-UART

Windows 10 WDF Driver Documentation

Developed with Windows Driver Foundation (WDF) Kernel-Mode Driver Framework (KMDF)

Ver1.19

Revision 01p2 10/1/21
Corresponding Hardware: Revision 02+
PMC 10-2015-0601
FLASH 0210

PMC-BiSerial-VI-UART
WDF Device Drivers

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

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.

This product has been designed to operate with PMC carriers and compatible user-provided equipment. Connection of incompatible hardware is likely to cause serious damage.

Table of Contents

Introduction

PmcBis6Uart is an 8 UART port PMC compatible interface card. This driver was developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The UART functionality is implemented in a Xilinx FPGA. It implements a PCI interface, FIFO's and protocol control/status for 8 channels. Each channel has separate 255 x 32 bit receive data and transmit data FIFO's.

When the PmcBis6Uart board is recognized by the PCI bus configuration utility it will load the PmcBis6Uart driver which will create a device object for the board, initialize the hardware, and create child devices for the 8 I/O channels.

Software Description

The PmcBis6Uart driver supports simultaneous operation of all ports independently. The driver and HW support both a packed and non-packed mode of operation. Non-packed mode functions as a virtual 8-bit port simulating the standard UART mode of operation. Specifically, each access to the read/write port transfers 1 byte of data.

Packed mode supports 4 bytes of data per access. This mode can be controlled via the IOCTL_UART_SET_CHANNEL_CONFIG. Tx access and Rx access can be set independently of one another.

Driver Installation

There are several files provided in each driver package. These files include UartBasePublic.h, pmcbis6uart_base.inf, pmcbis6uart_base.cat, pmcbis6uart_base.sys, UartChanPublic.h, UartChanb.inf, uartchan.cat, UartChan.sys.

UartBasePublic.h and UartChanPublic.h are the C header file that define the Application Program Interface (API) for the PmcBis6Uart drivers. 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 pmcbis6uart_base.inf, pmcbis6uart_base.cat, pmcbis6uart_base.sys, and to an easy to navigate to directory.

With the PMC-BISERIAL-VI-UART hardware installed, power-on the PCI host computer.

- Open the ***Device Manager*** from the control panel (or use search to find it).
- 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 **Browse..** and navigate to the folder where you previously moved the files.
- Select the folder and click **OK**.
- Click **Next**.
- Select ***Close*** to close the update window.

The system should now display the UartBase PCI adapter in the Device Manager and it will now display the 8 channel devices as well.

Follow the same steps for the 1st channel device, and the following ones you can tell windows to search for the device automatically (the first time puts the driver in the "windows store" which is why windows can find the driver for subsequent channels).

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 UartBasePublic.h and UartChanPublic.h. See main.c in the PmcBis6UartUserApp project for an example of how to acquire a handle to the device.

Note: In order to build an application you must link with setupapi.lib.

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 6 UART driver are described below:

IOCTL_UART_BASE_GET_INFO

Function: Returns the device driver version, design version, design type, user switch value, device instance number and PLL device ID.
Input: None
Output: PUART_BASE_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 UART_BASE_DRIVER_DEVICE_INFO below. Bit definitions can be found in the 'BASE_GP' section under Register Definitions in the Hardware manual.

```
typedef struct _UART_BASE_DRIVER_DEVICE_INFO
{
   UCHAR    DriverVersion;
   UCHAR    XilType;
   UCHAR    RevMaj;
   UCHAR    RevMin;
   UCHAR    PllDeviceId;
   UCHAR    SwitchValue;
   ULONG    InstanceNumber;
```

```
} UART_BASE_DRIVER_DEVICE_INFO,
*PUART_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_UART_BASE_GET_STATUS

Function: Returns Interrupt Base Status Register.
Input: None
Output: ULONG
Notes: Provides the interrupt status of each of the 8 channels. Bit definitions can be found in 'BASE_INT' section under Register Definitions in the Hardware manual.

IOCTL_UART_BASE_LOAD_PLL

Function: Loads the internal registers of the PLL.
Input: UART_BASE_PLL_DATA structure
Output: None
Notes: After the PLL has been configured, the register array data is analysed to determine the programmed frequencies, and the IO clock A-D initial divisor fields in the base control register are automatically updated.

IOCTL_UART_BASE_READ_PLL

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

IOCTL_UART_CHAN_GET_INFO

Function: Returns the device driver version and instance number.
Input: None
Output: UART_CHAN_DRIVER_DEVICE_INFO structure
Notes: Instance number is the zero-based device number. See the definition of UART_CHAN_DRIVER_DEVICE_INFO below.

```
typedef struct _UART_CHAN_DRIVER_DEVICE_INFO {
   UCHAR    DriverVersion;
   ULONG    InstanceNumber;
} UART_CHAN_DRIVER_DEVICE_INFO,
*PUART_CHAN_DRIVER_DEVICE_INFO;
```

IOCTL_UART_CHAN_SET_CONT

Function: Specifies the base control configuration.
Input: UART_CHAN_CONT structure
Output: None
Notes: All bits are active high and are reset on system power up or reset. See the definition of UART_CHAN_CONT below. Bit definitions can be found in the 'UART_CHAN_CONT' section under Register Definitions in the Hardware manual.

```
typedef struct _UART_CHAN_CONT {
   BOOLEAN     lb_enable;
   BOOLEAN     tx_enable;
   BOOLEAN     rx_enable;
   BOOLEAN     rx_err_int_en;
   BOOLEAN     tx_fifo_amt_int_en;
   BOOLEAN     rx_fifo_afl_int_en;
   BOOLEAN     rx_ovrflow_int_en;
   BOOLEAN     rx_pkt_lvl_int_en;
   BOOLEAN     tx_break;
   BOOLEAN     tx_par_en;
   BOOLEAN     tx_par_odd;
   BOOLEAN     tx_stop_2;
   BOOLEAN     tx_len_8;
   BOOLEAN     rx_par_en;
   BOOLEAN     rx_par_odd;
   BOOLEAN     rx_stop_2;
   BOOLEAN     rx_len_8;
   BOOLEAN     tx_par_lvl;
   BOOLEAN     rx_par_lvl;
   TX_RX_MODE  tx_mode;
   TX_RX_MODE  rx_mode;
} UART_CHAN_CONT, *PUART_CHAN_CONT;

typedef enum _TX_RX_MODE {
   ONE_BYTE,
   PACKED,
   PACKETIZED,
   ALT_PACK,
   TEST,       // only valid for tx mode
} TX_RX_MODE, *PTX_RX_MODE;
```

IOCTL_UART_CHAN_GET_CONT

Function: Returns the fields set in the previous call.
Input: None
Output: UART_CHAN_CONT structure
Notes: Returns the values set in the previous call. See the definition of UART_CHAN_CONT above.

IOCTL_UART_CHAN_SET_CONT_B

Function: Specifies the base control configuration.
Input: UART_CHAN_CONT_B structure
Output: None
Notes: All bits are active high and are reset on system power up or reset. See the definition of UART_CHAN_CONT_B below. Bit definitions can be found in the 'UART_CHAN_CONTB' section under Register Definitions in the Hardware manual.

```
typedef struct _UART_CHAN_CONT_B {
   BOOLEAN           brk_rise_int_en;
   BOOLEAN           brk_fall_int_en;
   BOOLEAN           brk_int_en;
   BOOLEAN           tx_pck_done_int_en;
   BOOLEAN           dir_tx;
   BOOLEAN           term_rx;
   BOOLEAN           term_tx;
   BOOLEAN           rx_pck_done_int_en;
   UCHAR             tx_pck_delay_mask;
   BOOLEAN           tx_timer_en;
   BOOLEAN           timer_int_en;
   BOOLEAN           tx_timer_emsk;
   UART_TIMER_MODE   timer_mode;
   BOOLEAN           dir_rts;
   BOOLEAN           force_rts;
   BOOLEAN           inv_flow_cont;
   BOOLEAN           use_cts;
   BOOLEAN           term_rts;
   BOOLEAN           term_cts;
   BOOLEAN           pll_input;
} UART_CHAN_CONT_B, *PUART_CHAN_CONT_B;

typedef enum _UART_TIMER_MODE {
   DISABLE_BOTH,
   ENABLE_TIMER,
   ENABLE_TRISTATE,
   ENABLE_BOTH
} UART_TIMER_MODE, *PUART_TIMER_MODE;
```

IOCTL_UART_CHAN_GET_CONT_B

Function: Returns the fields set in the previous call.
Input: None

Output: UART_CHAN_CONT_B structure
Notes: Returns the values set in the previous call. See the definition of UART_CHAN_CONT_B above.

IOCTL_UART_CHAN_GET_STATUS

Function: Returns the value of the channel status register.
Input: None
Output: ULONG
Notes: See Channel status bit definitions below. You can use any of the Masks provided in the UartChanPublic.h file to mask off the desired bits. Bit definitions can be found in the 'UART_CHAN_STAT' section under Register Definitions in the Hardware manual.

```
// Channel Status bit definitions
#define STAT_TX_FF_MT             0x00000001
#define STAT_TX_FF_AMT            0x00000002
#define STAT_TX_FF_FL             0x00000004
#define STAT_TX_TIMER_LAT         0x00000008
#define STAT_RX_FF_MT             0x00000010
#define STAT_RX_FF_AFL            0x00000020
#define STAT_RX_FF_FL             0x00000040
#define STAT_RTS_STAT             0x00000080
#define STAT_TX_PAR_ERR_LAT       0x00000100
#define STAT_RX_FRM_ERR_LAT       0x00000200
#define STAT_RX_OVRFL_LAT         0x00000400
#define STAT_RX_LEN_OVRFL_LAT     0x00000800
#define STAT_WR_DMA_ERR           0x00001000
#define STAT_RD_DMA_ERR           0x00002000
#define STAT_WR_DMA_INT           0x00004000
#define STAT_RD_DMA_INT           0x00008000
#define STAT_RX_PCKT_FF_MT        0x00010000
#define STAT_RX_PCKT_FF_FL        0x00020000
#define STAT_TX_PCKT_FF_MT        0x00040000
#define STAT_TX_PCKT_FF_FL        0x00080000
#define STAT_LOC_INT              0x00100000
#define STAT_INT_STAT             0x00200000
#define STAT_RX_PCKT_DONE_LAT     0x00400000
#define STAT_TX_PCKT_DONE_LAT     0x00800000
#define STAT_TX_IDLE              0x01000000
#define STAT_RX_IDLE              0x02000000
#define STAT_BURST_IN_IDLE        0x04000000
#define STAT_BURST_OUT_IDLE       0x08000000
#define STAT_BRK_STAT_LAT         0x10000000
#define STAT_BRK_STAT             0x20000000
#define STAT_TX_AMT_LAT           0x40000000
#define STAT_RX_AFL_LAT           0x80000000
```

IOCTL_UART_CHAN_CLEAR_STATUS

Function: Clears specified latched status bits then returns the value of the channel status register.
Input: ULONG
Output: None
Notes: Write to the bit to clear the specific latch to be cleared. . Bit definitions can be found in the 'UART_CHAN_STAT' section under Register Definitions in the Hardware manual.

IOCTL_UART_CHAN_SET_BAUD_RATE

Function: Write to set TX/RX baud rate.
Input: UART_CHAN_BAUD_RATE
Output: None
Notes: See the definition of UART_CHAN_BAUD_RATE below. Definition can be found in the 'CHAN_BAUD_RATE' section under Register Definitions in the Hardware manual.

```
typedef struct _UART_CHAN_BAUD_RATE{
   USHORT   TxBaudRate;
   USHORT   RxBaudRate;
} UART_CHAN_BAUD_RATE, *PUART_CHAN_BAUD_RATE;
```

IOCTL_UART_CHAN_GET_BAUD_RATE

Function: Read to get TX/RX baud rate
Input: None
Output: UART_CHAN_BAUD_RATE
Notes: Returns the values set in the previous call. See the definition of UART_CHAN_BAUD_RATE above.

IOCTL_UART_CHAN_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: UART_CHAN_FIFO_LEVELS structure
Output: None
Notes: Almost empty and Almost full should be set to 0x0010 and 0x00EF respectively before use of FIFOS. The FIFO counts are compared to these levels to set the value of the CHAN_STAT_TX_FF_AMT and CHAN_STAT_RX_FF_AFL status bits and latch the CHAN_STAT_TX_AMT_LT and CHAN_STAT_RX_AFL_LT latched status bits. See the definition of UART_CHAN_FIFO_LEVELS below. Full definition can be found in the

‘CHAN_TXFIFO_LVL’ and the ‘CHAN_RXFIFO_LVL’ sections under Register Definitions in the Hardware manual.

```
typedef struct _UART_CHAN_FIFO_LEVELS {
   USHORT   AlmostFull;
   USHORT   AlmostEmpty;
} UART_CHAN_FIFO_LEVELS, *PUART_CHAN_FIFO_LEVELS;
```

IOCTL_UART_CHAN_GET_FIFO_LEVELS

Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: UART_CHAN_FIFO_LEVELS structure
Notes: Returns the values set in the previous call. See the definition of UART_CHAN_FIFO_LEVELS above.

IOCTL_UART_CHAN_SET_FRAME_TIME

Function: Write to set Frame time
Input: ULONG
Output:
Notes: Programmable count to determine how long to wait without a new character arriving for receiver to declare “end of packet”. Full definition can be found under Register definitions under CHAN_FRAME_TIME in hardware manual

IOCTL_UART_CHAN_GET_FRAME_TIME

Function: Read to get Frame time
Input: None
Output: ULONG

IOCTL_UART_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive data and packet-length FIFOs.
Input: None
Output: UART_CHAN_FIFO_COUNTS structure
Notes: The FIFOs are both 256 deep. See the definition of UART_CHAN_FIFO_COUNTS below. Full definition can be found in the ‘CHAN_RX_FIFO_CNT’ AND ‘CHAN_TX_FIFO_CNT’ sections under Register Definitions in the Hardware manual.

```
typedef struct _UART_CHAN_FIFO_COUNTS {
   USHORT   TxDataCnt;
   USHORT   TxPktCnt;
   USHORT   RxDataCnt;
   USHORT   RxPktCnt;
} UART_CHAN_FIFO_COUNTS, *PUART_CHAN_FIFO_COUNTS;
```

IOCTL_UART_CHAN_RESET_FIFOS

Function: Resets TX and/or RX FIFOs for specified channel.
Input: UART_FIFO_SEL
Output: None
Notes: Call the function with UART_TX, UART_RX, or UART_BOTH to reset the desired FIFO. See Definition of UART_FIFO_SEL below.

```
typedef enum _UART_FIFO_SEL {
   UART_TX,
   UART_RX,
   UART_BOTH
} UART_FIFO_SEL, *PUART_FIFO_SEL;
```

IOCTL_UART_CHAN_REGISTER_EVENT

Function: Registers an event to be signaled when an interrupt occurs.
Input: Handle to the Event object
Output: None
Notes: The caller creates an event with CreateEvent() and supplies the handle returned from that call as the input to this IOCTL. The driver then obtains a system pointer to the event and signals the event when a user interrupt is serviced. The user interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_UART_CHAN_ENABLE_INTERRUPT

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

IOCTL_UART_CHAN_DISABLE_INTERRUPT

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

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

IOCTL_UART_CHAN_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 new field is true if the Status has been updated since it was last read.

IOCTL_UART_CHAN_SWW_TX_FIFO

Function: Writes a single longword to TX FIFO.
Input: Data (unsigned long)
Output: None
Notes: Data is the longword to write. Full definition can be found in the 'CHAN_UART_FIFO' section under Register Definitions in the Hardware manual.

IOCTL_UART_CHAN_SWR_RX_FIFO

Function: Reads a single longword from RX FIFO.
Input: None
Output: Data (unsigned long)
Notes: Read data is the one written in above IOCTL.

IOCTL_UART_CHAN_WRITE_PKT_LEN

Function: Write a received packet-length value from the packet-length FIFO.
Input: PUSHORT
Output: None
Notes: Full definition can be found in the 'CHAN_PACKET_FIFO' section under Register Definitions in the Hardware manual.

IOCTL_UART_CHAN_READ_PKT_LEN

Function: Reads a received packet-length value from the packet-length FIFO.
Input: None
Output: UART_PACKET_FIFO
Notes: UART_PACKET_FIFO includes parity errors, frame errors, Rx overflow errors or Rx length overflow errors that occur.

```
typedef struct _UART_PACKET_FIFO {
   USHORT      RX_PKT_FIFO;
   BOOLEAN     ParErr;
   BOOLEAN     FrmErr;
   BOOLEAN     RxDataOvflErr;
   BOOLEAN     RxPckOvflErr;
} UART_PACKET_FIFO, *PUART_PACKET_FIFO;
```

IOCTL_UART_CHAN_SET_TIMER

Function: Write to set Timer register
Input: ULONG
Output:
Notes: Programmable count to define a range used in the TxTimer32 function. Full definition can be found in the Register definitions under CHAN_TX_TIMER_MOD in hardware manual

IOCTL_UART_CHAN_GET_TIMER

Function: Read from Timer register
Input: None
Output: ULONG
Notes: Reads back the value written in the Timer register

IOCTL_UART_CHAN_GET_TIMER_CNT

Function: Read from Timer Count register.
Input: None
Output: ULONG
Notes: Allows user to monitor the current count in the TxTimer32 function

Write

PmcBis6Uart RAM data is written to the device using the write command. Writes are executed using the 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

PmcBis6Uart RAM data is read from the device using the read command. Reads are executed using the 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.

For PmcBis6Uart write and read are implemented with Kernel level write and read for high performance.

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