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

XmBase XmChan AtpVtx TstVtx GenVtx

Driver Documentation

Windows Driver Foundation

Revision A
Corresponding Hardware: Revision B/C
10-2007-0201
Corresponding Firmware: Revision H

XmBase, XmChan, AtpVtx, TstVtx and GenVtx
WDF Device Drivers for the
PMC-XM-Diff - PMC based interface module With Re-programmable I/O logic

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

Table of Contents

Introduction

The XmBase and XmChan drivers are Windows device drivers for the PMC-XM from Dynamic Engineering. These drivers were developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The PMC-XM board has a Spartan3-2000 Xilinx FPGA to implement the PCI interface, two input and two output scatter-gather DMA engines with 4k x 32-bit data FIFOs for each. There is also a Virtex4- SX35/LX60 Xilinx that is programmed from an on-board flash PROM on power-up, but can be re-configured from a bit-file through the PCI interface if desired.

The AtpVtx and TstVtx drivers, used by Dynamic Engineering to test the PMC-XM, control the Virtex designs that are supplied with the board. The GenVtx driver is a generic driver that explicitly does some basic services: controlling LEDs, interfacing with the on-board PLL and handling interrupts, but otherwise only reads or writes data from a specified address offset. This allows the user to control their custom Virtex design without having a driver specific to that design. If a custom driver is desired, contact Dynamic Engineering and we can write a driver to match your specifications.

There is a field in the Virtex Base Status Register to specify the design ID and revision. This is read by the XmBase driver to determine which Virtex driver to load. The AtpVtx driver is assigned to design ID = 0, EnMbVtx design ID = 1 and the TstVtx design ID = 0x55. The TstVtx driver is used to test loading a Virtex design from a bit-file over the PCI bus and to verify the accuracy of the PLL clocks.

Currently any other ID number will load the generic driver, but altering the XmVirtex.inf file will allow other drivers to be assigned to other design IDs. This makes it possible for the I/O functionality to be changed on the fly by reloading the Virtex from a bit-file after which the base driver will re-read the Virtex design ID and automatically load the appropriate Virtex driver.

When the PMC-XM is recognized by the PCI bus configuration utility it will start the XmBase driver. The XmBase driver enumerates the channels and creates two separate XmChan device objects. This allows the I/O channels to be totally independent while the base driver controls the device items that are common. A Virtex device object will also be created at this time based on the design ID as described above. 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 I/O channel devices using scatter-gather DMA.

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-XM user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. The files needed to install the drivers are XmBase.inf, XmBase.cat, XmBase.sys, XmChan.inf, XmChan.cat, XmChan.sys, XmVirtex.inf, XmVirtex.cat, AtpVtx.sys, TstVtx.sys, GenVtx.sys and WdfCoInstaller01009.dll.

XmBasePublic.h, XmChanPublic.h, AtpVtxPublic.h, TstVtxPublic.h and GenVtxPublic.h are C header files that define the Application Program Interface (API) to the drivers. These files are required at compile time by any application that wishes to interface with the drivers, but they are not needed for driver installation.

Windows XP Installation

Copy XmBase.inf, XmBase.cat, XmBase.sys, XmChan.inf, XmChan.cat, XmChan.sys, XmVirtex.inf, XmVirtex.cat and AtpVtx.sys, TstVtx.sys, GenVtx.sys and WdfCoInstaller01009.dll (XP version) to a floppy disk, CD or USB memory device as preferred.

With the PMC-XM hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.

- Insert the disk or memory device prepared above in the desired drive.
- Select ***No*** when asked to connect to Windows Update.
- Select ***Next***.
- Select ***Install the software automatically***. (If not found go to the next line)
- Select ***Install the software from a specific location***. (Specify your file's location)
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

The system should now see the PMC-XM I/O channels (and possibly the Virtex device) and reopen the ***New Hardware Wizard***. Proceed as above for each device as necessary.

If the Virtex is not seen it may be necessary to restart the host computer to load registry information.

Windows 7 Installation

Copy XmBase.inf, XmBase.cat, XmBase.sys, XmChan.inf, XmChan.cat, XmChan.sys, XmVirtex.inf, XmVirtex.cat and AtpVtx.sys, TstVtx.sys, GenVtx.sys and WdfCoInstaller01009.dll (Win7 version) to a floppy disk, CD or USB memory device as preferred.

With the PMC-XM 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 PMC-XM I/O channels (and possibly the Virtex device) in the Device Manager.

- Right-click on each device icon, select ***Update Driver Software*** and proceed as above for each device as necessary. If the Virtex is not seen it may be necessary to restart the host computer to load registry information.

***** 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 drivers have been installed they will start automatically when the system recognizes the hardware.

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

The interfaces to the devices are identified using globally unique identifiers (GUID), which are defined in XmBasePublic.h, XmChanPublic.h, AtpVtxPublic.h, TstVtxPublic.h and GenVtxPublic.h. See main.c in the PmcXmUserApp project for an example of how to acquire handles for the base, two channel devices and the Virtex.

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 devices. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (see below), 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 XmBase driver are described below:

IOCTL_XM_BASE_GET_INFO

Function: Returns the Driver revision, Xilinx flash revision, Switch value, and Instance number.
Input: None
Output: XM_BASE_DRIVER_DEVICE_INFO structure
Notes: Switch value is the configuration of the onboard dip-switch 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 XM_BASE_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _XM_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    XilinxRev;
   UCHAR    SwitchValue;
   ULONG    InstanceNum;
} XM_BASE_DRIVER_DEVICE_INFO, *PXM_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_XM_BASE_SET_CONFIG

Function: Sets the value of the base control register.
Input: XM_BASE_CONFIG structure
Output: None
Notes: The JtagOutEn function is not currently implemented. See the definitions of XM_FLASH_SEL and XM_BASE_CONFIG below.

```
 // Selects which device's Flash is written by JTAG
typedef enum _XM_FLASH_SEL {
   EXT_SEL,     // Flash device controled by external strap
   S3_SEL,      // Spartan 3 Flash selected
   VTX_SEL      // Virtex 4 Flash selected
} XM_FLASH_SEL, *PXM_FLASH_SEL;

typedef struct _XM_BASE_CONFIG {
   XM_FLASH_SEL   FlashSel;   // Routes JTAG to Flash devices
   BOOLEAN        VtxCtlRst;  // Resets the Virtex access control module
   BOOLEAN        JtagOutEn;  // Enables Spartan to drive local JTAG signals
} XM_BASE_CONFIG, *PXM_BASE_CONFIG;
```

IOCTL_XM_BASE_GET_CONFIG

Function: Returns the value of the base control register.
Input: None
Output: XM_BASE_CONFIG structure
Notes: Reads and returns the fields of the structure above.

IOCTL_XM_BASE_GET_STATUS

Function: Returns the value of the base status register.
Input: None
Output: Value of the base status register (unsigned long integer)
Notes: See the status bit definitions below.

```
 // Status bit definitions
#define STATUS_LOCAL_INT                 0x00000001
#define STATUS_CHAN0_INT                 0x00000010
#define STATUS_CHAN1_INT                 0x00000020
#define STATUS_VIRTEX_INT0               0x00000040
#define STATUS_VIRTEX_INT1               0x00000080
#define STATUS_VIRTEX_DONE               0x00000100
#define STATUS_VIRTEX_INIT               0x00000200
#define STATUS_VIRTEX_PROG               0x00000400
#define STATUS_VIRTEX_STAT0              0x00100000
#define STATUS_VIRTEX_STAT1              0x00200000
#define STATUS_VIRTEX_ACK                0x01000000
#define STATUS_INT_ACTIVE                0x80000000
```

IOCTL_XM_BASE_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's interrupt service routine waits on this event, allowing it to respond to the interrupt.

IOCTL_XM_BASE_ENABLE_INTERRUPT

Function: Enables the master interrupt.
Input: None
Output: None
Notes: Interrupts will be enabled when the device initializes. This command is run to re-enable interrupt processing if interrupts were previously disabled.

IOCTL_XM_BASE_DISABLE_INTERRUPT

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

IOCTL_XM_BASE_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 test and development, to test interrupt processing.

IOCTL_XM_BASE_GET_ISR_STATUS

Function: Returns the interrupt status that was 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 for the last interrupt serviced. See the list of status bits defined above following the GET_STATUS call description.

IOCTL_XM_BASE_LOAD_VIRTEX

Function: Reloads the Virtex from a specified bit-file.
Input: The name of the bit-file (VIRTEX_LOAD structure)
Output: None
Notes: In order for the driver to find the Virtex bit-file, it must reside in a folder named VirtexDesigns within the WINDOWS folder (specified in XmBase.inf). See the definition of VIRTEX_LOAD below.

```
#define XM_BASE_FILE_NAME_SZ      40

typedef struct _VIRTEX_LOAD {
   WCHAR    FileName[XM_BASE_FILE_NAME_SZ];
} VIRTEX_LOAD, *PVIRTEX_LOAD;
```

The IOCTLs defined for the XmChan driver are described below:

IOCTL_XM_CHAN_GET_INFO

Function: Returns the Driver revision and Instance number.
Input: None
Output: XM_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of XM_CHAN_DRIVER_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _XM_CHAN_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   ULONG    InstanceNum;
} XM_CHAN_DRIVER_DEVICE_INFO, *PXM_CHAN_DRIVER_DEVICE_INFO;
```

IOCTL_XM_CHAN_SET_CONFIG

Function: Writes to the channel's control register.
Input: XM_CHAN_CONFIG structure
Output: None
Notes: Specifies the FIFO loopback enable, transfer enables to and from the Virtex, DMA preemption behavior and enabled interrupt sources. See the definitions of XM_DMA_PRMPT and XM_CHAN_CONFIG below.

```
 // Channel DMA priority (use sparingly)
typedef enum _XM_DMA_PRMPT {
   XM_NONE,     // No priority
   XM_READ,     // Read DMA has priority
   XM_WRITE,    // Write DMA has priority
   XM_RDWR      // Read and Write DMA have priority (for this channel)
} XM_DMA_PRMPT, *PXM_DMA_PRMPT;

typedef struct _XM_CHAN_CONFIG {
   BOOLEAN         FifoTest;
   BOOLEAN         TxEnable;
   BOOLEAN         RxEnable;
   BOOLEAN         TxAmtIntEn;
   BOOLEAN         RxAflIntEn;
   BOOLEAN         VirtexIntEn;
   XM_DMA_PRMPT    DmaPriority;
} XM_CHAN_CONFIG, *PXM_CHAN_CONFIG;
```

IOCTL_XM_CHAN_GET_CONFIG

Function: Returns the configuration of the control register.
Input: None
Output: XM_CHAN_CONFIG structure
Notes: Reads and returns the fields of the structure above. This command is used mainly for testing.

IOCTL_XM_CHAN_GET_STATUS

Function: Returns the channel's status value and clears the latched bits.
Input: None
Output: Value of the channel's status register (unsigned long integer)
Notes: The latched almost empty and almost full and the read and write DMA error bits are the only latched bits cleared by this call, the DMA interrupt status bits are cleared in the DMA interrupt service routines. See the status bit definitions below.

```
 // Status bit definitions
#define CHAN_STAT_TX_FF_MT      0x00000001
#define CHAN_STAT_TX_FF_AMT     0x00000002
#define CHAN_STAT_TX_FF_FL      0x00000004
#define CHAN_STAT_TX_FF_VLD     0x00000008
#define CHAN_STAT_RX_FF_MT      0x00000010
#define CHAN_STAT_RX_FF_AFL     0x00000020
#define CHAN_STAT_RX_FF_FL      0x00000040
#define CHAN_STAT_RX_FF_VLD     0x00000080
#define CHAN_STAT_TX_AMT_LAT    0x00000100
#define CHAN_STAT_RX_AFL_LAT    0x00000200
#define CHAN_STAT_WR_DMA_ERR    0x00001000
#define CHAN_STAT_RD_DMA_ERR    0x00002000
#define CHAN_STAT_WR_DMA_INT    0x00004000
#define CHAN_STAT_RD_DMA_INT    0x00008000
#define CHAN_STAT_LOC_INT       0x00010000
#define CHAN_STAT_VIRTEX_INT    0x00020000
#define CHAN_STAT_WR_DMA_RDY    0x00040000
#define CHAN_STAT_RD_DMA_RDY    0x00080000
#define CHAN_STAT_INTSTAT       0x80000000
```

IOCTL_XM_CHAN_RESET_FIFOS

Function: Resets the channel's TX and/or RX FIFOs.
Input: FIFO(s) to reset (XM_FIFO_SEL enumerated type)
Output: None
Notes: Resets the TX and/or RX FIFOs and all the associated data registers and state-machines for the referenced channel. See the definition of XM_FIFO_SEL below.

```
typedef enum _XM_FIFO_SEL {
   XM_TX,
   XM_RX,
   XM_BOTH
} XM_FIFO_SEL, *PXM_FIFO_SEL;
```

IOCTL_XM_CHAN_SET_FIFO_LEVELS

Function: Sets the channel's transmitter FIFO almost empty and receiver FIFO almost full levels.
Input: XM_CHAN_FIFO_LEVELS structure
Output: None
Notes: The FIFO levels are used to set the threshold for the transmit FIFO almost empty and receive FIFO almost full status bits. The value represents the number of 32-bit data-words in the corresponding FIFO that causes the relevant status bit to change states. This level is also used to determine when DMA preemption is applied, if enabled. See the definition of XM_CHAN_FIFO_LEVELS below.

```
typedef struct _XM_CHAN_FIFO_LEVELS {
   USHORT   AlmostFull;
   USHORT   AlmostEmpty;
} XM_CHAN_FIFO_LEVELS, *PXM_CHAN_FIFO_LEVELS;
```

IOCTL_XM_CHAN_GET_FIFO_LEVELS

Function: Returns the channel's transmitter FIFO almost empty and receiver FIFO almost full levels.
Input: None
Output: XM_CHAN_FIFO_LEVELS structure
Notes: See the definition of XM_CHAN_FIFO_LEVELS above.

IOCTL_XM_CHAN_WRITE_FIFO

Function: Writes a data word to the channel's transmit FIFO.
Input: FIFO data word (unsigned long integer)
Output: None
Notes: This call writes a single 32-bit word to the FIFO regardless of FIFO state. If the FIFO is already full, the data word is lost.

IOCTL_XM_CHAN_READ_FIFO

Function: Reads a data word from the channel's receive FIFO.
Input: None
Output: FIFO data word (unsigned long integer)
Notes: As with the previous call, the FIFO state is not checked when this operation is performed. If the FIFO was already empty, the last word in the FIFO will be returned repeatedly.

IOCTL_XM_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmitter and receiver FIFOs.
Input: None
Output: XM_CHAN_FIFO_COUNTS structure
Notes: The values returned by this call include the data-words in the receive data pipeline. The CHAN_STAT_RX_FF_VLD status bit indicates when there is valid data even though the CHAN_STAT_RX_FF_MT status bit indicates that the FIFO is empty. See the definition of XM_CHAN_FIFO_COUNTS below.

```
typedef struct _XM_CHAN_FIFO_COUNTS {
   USHORT   TxCount;
   USHORT   RxCount;
} XM_CHAN_FIFO_COUNTS, *PXM_CHAN_FIFO_COUNTS;
```

IOCTL_XM_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. The DMA interrupts do not cause the event to be signaled. To un-register the event, set the input handle to NULL.

IOCTL_XM_CHAN_ENABLE_INTERRUPT

Function: Enables the channel master interrupt.
Input: None
Output: None
Notes: Interrupts will be enabled when the device initializes. When servicing a user interrupt, the channel master interrupt is disabled in the driver interrupt service routine. This command must be run after each user interrupt occurs to re-enable interrupt processing.

IOCTL_XM_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_XM_CHAN_FORCE_INTERRUPT

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

IOCTL_XM_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: XM_CHAN_ISR_STAT structure
Notes: Returns the interrupt status that was read in the interrupt service routine for the last user interrupt serviced and a Boolean field that indicates whether this status has been updated since it was last read. The DMA interrupts do not update this value. See the definition of XM_CHAN_ISR_STAT below. The status bits returned in the Status field are listed after the IOCTL_XM_CHAN_GET_STATUS call above.

```
typedef struct _XM_CHAN_ISR_STAT {
   ULONG    Status;
   BOOLEAN  New;
} XM_CHAN_ISR_STAT, *PXM_CHAN_ISR_STAT;
```

The IOCTLs defined for the AtpVtx driver are described below:

IOCTL_ATP_VTX_GET_INFO

Function: Returns the Design ID and revision, Driver revision, Instance number and PLL device ID.
Input: None
Output: ATP_VTX_DRIVER_DEVICE_INFO structure
Notes: 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 ATP_VTX_DDINFO below.

```
typedef struct _ATP_VTX_DDINFO {
   UCHAR    DriverRev;
   UCHAR    PllDeviceId;
   UCHAR    DesignId;
   UCHAR    DesignRev;
   ULONG    InstanceNum;
} ATP_VTX_DDINFO, *PATP_VTX_DDINFO;
```

IOCTL_ATP_VTX_SET_BASE_CONFIG

Function: Writes the base control register configuration for the Virtex ATP design.
Input: ATP_VTX_BASE_CONFIG structure
Output: None
Notes: The LEDs are lit when the corresponding field is TRUE. IoEnable enables the I/O subsystem and IoMuxSel determines which ports are the source and destination of the I/O data and clock. ResetDcm does a manual reset of the Digital Clock Manager. See the definitions of IO_MUX_SEL and ATP_VTX_BASE_CONFIG below.

```
typedef enum _IO_MUX_SEL {
   ATP_SEL_0,  // chan0-A -> chan0-A, chan0-B -> chan0-B, clk0 IO32 -> IO33 clk0
   ATP_SEL_1,  // chan0-A -> chan0-A, chan0-B -> chan0-B, clk0 IO33 -> IO32 clk0
   ATP_SEL_2,  // chan1-A -> chan1-A, chan1-B -> chan1-B, clk1 IO32 -> IO33 clk1
   ATP_SEL_3,  // chan1-A -> chan1-A, chan1-B -> chan1-B, clk1 IO33 -> IO32 clk1
   ATP_SEL_4,  // chan0-A -> chan0-B, chan0-B -> chan0-A, clk0 IO32 -> IO33 clk0
   ATP_SEL_5,  // chan0-A -> chan0-B, chan0-B -> chan0-A, clk0 IO33 -> IO32 clk0
   ATP_SEL_6,  // chan1-A -> chan1-B, chan1-B -> chan1-A, clk1 IO32 -> IO33 clk1
   ATP_SEL_7,  // chan1-A -> chan1-B, chan1-B -> chan1-A, clk1 IO33 -> IO32 clk1
   ATP_SEL_8,  // chan0-A -> chan1-A, chan0-B -> chan1-B, clk0 IO32 -> IO33 clk1
   ATP_SEL_9,  // chan0-A -> chan1-A, chan0-B -> chan1-B, clk0 IO33 -> IO32 clk1
   ATP_SEL_A,  // chan1-A -> chan0-A, chan1-B -> chan0-B, clk1 IO32 -> IO33 clk0
   ATP_SEL_B,  // chan1-A -> chan0-A, chan1-B -> chan0-B, clk1 IO33 -> IO32 clk0
   ATP_SEL_C,  // chan0-A -> chan1-B, chan0-B -> chan1-A, clk0 IO32 -> IO33 clk1
   ATP_SEL_D,  // chan0-A -> chan1-B, chan0-B -> chan1-A, clk0 IO33 -> IO32 clk1
   ATP_SEL_E,  // chan1-A -> chan0-B, chan1-B -> chan0-A, clk1 IO32 -> IO33 clk0
   ATP_SEL_F   // chan1-A -> chan0-B, chan1-B -> chan0-A, clk1 IO33 -> IO32 clk0
} IO_MUX_SEL, *PIO_MUX_SEL;
```

```
typedef struct _ATP_VTX_BASE_CONFIG {
   BOOLEAN     Led0;
   BOOLEAN     Led1;
   BOOLEAN     Led2;
   BOOLEAN     Led3;
   IO_MUX_SEL  IoMuxSel;
   BOOLEAN     IoEnable;
   BOOLEAN     ResetDcm;
} ATP_VTX_BASE_CONFIG, *PATP_VTX_BASE_CONFIG;
```

IOCTL_ATP_VTX_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: ATP_VTX_BASE_CONFIG structure
Notes: Returns the values set in the preceding call.

IOCTL_ATP_VTX_GET_BASE_STATUS

Function: Returns the base status register value.
Input: None
Output: ATP_VTX_BASE_STATUS structure
Notes: See the definition of ATP_VTX_BASE_STATUS below.

```
typedef struct _ATP_VTX_BASE_STATUS {
   BOOLEAN     Chan0IntActv;
   BOOLEAN     Chan1IntActv;
} ATP_VTX_BASE_STATUS, *PATP_VTX_BASE_STATUS;
```

IOCTL_ATP_VTX_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: ATP_VTX_PLL_DATA structure
Output: None
Notes: The ATP_VTX_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the data to write.

```
typedef struct _ATP_VTX_PLL_DATA {
   UCHAR    Data[PLL_MESSAGE_SIZE];
} ATP_VTX_PLL_DATA, *PATP_VTX_PLL_DATA;
```

IOCTL_ATP_VTX_READ_PLL_DATA

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

IOCTL_ATP_VTX_SET_CHAN_CONFIG

Function: Writes to a channel’s control register.
Input: Channel number and configuration value (ATP_VTX_CHAN_WRITE)
Output: None
Notes:

```
typedef struct _ATP_VTX_CHAN_CONFIG {
   UCHAR        Channel;
   BOOLEAN      FifoTest;
   BOOLEAN      TxEnable;
   BOOLEAN      RxEnable;
   BOOLEAN      TxIntEn;
   BOOLEAN      RxIntEn;
   BOOLEAN      TxPort;  // 0=>Tx port A, 1=>Tx port B
   BOOLEAN      RxPort;  // 0=>Rx port A, 1=>Rx port B
   USHORT       TxSndCnt;
} ATP_VTX_CHAN_CONFIG, *PATP_VTX_CHAN_CONFIG;
```

IOCTL_ATP_VTX_GET_CHAN_CONFIG

Function: Returns the configuration of the control register.
Input: Channel number (unsigned character)
Output: Value of control register (unsigned long integer)
Notes:

IOCTL_ATP_VTX_GET_CHAN_STATUS

Function: Returns the channel’s status value.
Input: Channel number (unsigned character)
Output: Value of the channel’s status register (unsigned long integer)
Notes: The TX_LAT and RX_LAT status bits will be returned and cleared, if set, when this call is made. See the status bit definitions below.

```
 // Channel status bit defines
#define ATP_VTX_CHAN_STAT_TX_MT     0x00000001
#define ATP_VTX_CHAN_STAT_TX_PMT    0x00000002
#define ATP_VTX_CHAN_STAT_TX_PFL    0x00000004
#define ATP_VTX_CHAN_STAT_TX_FL     0x00000008
#define ATP_VTX_CHAN_STAT_RX_MT     0x00000010
#define ATP_VTX_CHAN_STAT_RX_PMT    0x00000020
#define ATP_VTX_CHAN_STAT_RX_PFL    0x00000040
#define ATP_VTX_CHAN_STAT_RX_FL     0x00000080
#define ATP_VTX_CHAN_STAT_TX_VLD    0x00000100
#define ATP_VTX_CHAN_STAT_TX_LAT    0x00000200
#define ATP_VTX_CHAN_STAT_RX_LAT    0x00000400
#define ATP_VTX_CHAN_LOC_INT        0x00000800
#define ATP_VTX_CHAN_INT_STAT       0x00008000
#define ATP_VTX_CHAN_RX_CNT_MSK     0xFFFF0000
```

IOCTL_ATP_VTX_RESET_FIFO

Function: Resets a channel's FIFO.
Input: ATP_VTX_CHAN_SEL enumeration type
Output: None
Notes: Resets either the TX or RX FIFO for one of the two channels. See the definition of ATP_VTX_CHAN_SEL below.

```
typedef enum _ATP_VTX_CHAN_SEL {
   ATP_TX0,
   ATP_RX0,
   ATP_TX1,
   ATP_RX1
} ATP_VTX_CHAN_SEL, *PATP_VTX_CHAN_SEL;
```

IOCTL_ATP_VTX_WRITE_FIFO

Function: Writes a data word to a channel's transmit or receive FIFO.
Input: FIFO to write (ATP_VTX_CHAN_SEL) and FIFO data word (ATP_VTX_FIFO_WRITE)
Output: None
Notes: See the definition of ATP_VTX_FIFO_WRITE below.

```
typedef struct _ATP_VTX_FIFO_WRITE {
   ATP_VTX_CHAN_SEL  FifoSelect;
   ULONG             Data;
} ATP_VTX_FIFO_WRITE, *PATP_VTX_FIFO_WRITE;
```

IOCTL_ATP_VTX_READ_FIFO

Function: Reads a data word from a channel's transmit or receive FIFO.
Input: FIFO to read (ATP_VTX_CHAN_SEL)
Output: FIFO data word (unsigned long integer)
Notes: See the definition of ATP_VTX_CHAN_SEL after the RESET_FIFO call.

IOCTL_ATP_VTX_SET_FIFO_LEVELS

Function: Sets a channel's transmitter almost empty and receiver almost full levels.
Input: Channel number and FIFO levels (ATP_VTX_FIFO_LEVELS structure)
Output: None
Notes: The FIFO levels are used to determine status when the FIFO data counts reach the specified levels. See the definition of ATP_VTX_FIFO_LEVELS.

```
typedef struct _ATP_VTX_FIFO_LEVELS {
   UCHAR    Channel;
   USHORT   AlmostFull;
   USHORT   AlmostEmpty;
} ATP_VTX_FIFO_LEVELS, *PATP_VTX_FIFO_LEVELS;
```

IOCTL_ATP_VTX_GET_FIFO_LEVELS

Function: Returns a channel's transmitter almost empty and receiver almost full levels.
Input: Channel number (unsigned character)
Output: ATP_VTX_FIFO_LEVELS structure
Notes: See the definition of ATP_VTX_FIFO_LEVELS above.

IOCTL_ATP_VTX_GET_FIFO_COUNTS

Function: Returns the number of data words in a channel's transmit and receive FIFOs.
Input: Channel number (unsigned character)
Output: ATP_VTX_FIFO_COUNTS structure
Notes: See the definition of ATP_VTX_FIFO_COUNTS below.

```
typedef struct _ATP_VTX_FIFO_COUNTS {
   USHORT   TxWordCount;
   USHORT   RxWordCount;
} ATP_VTX_FIFO_COUNTS, *PATP_VTX_FIFO_COUNTS;
```

IOCTL_ATP_VTX_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_ATP_VTX_ENABLE_INTERRUPT

Function: Enables a channel's interrupt.
Input: Channel number (unsigned character)
Output: None
Notes: This command must be run to allow the driver to respond to local interrupts. The interrupt enable is disabled in the driver interrupt service routine. Therefore this command must be run after each interrupt occurs to re-enable it.

IOCTL_ATP_VTX_DISABLE_INTERRUPT

Function: Disables a channel's interrupt.
Input: Channel number (unsigned character)
Output: None
Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_ATP_VTX_FORCE_INTERRUPT

Function: Causes a channel’s interrupt to occur.
Input: Channel number (unsigned character)
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as the channel’s interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_ATP_VTX_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: Channel number (unsigned character)
Output: Interrupt status value (unsigned long integer)
Notes: Returns the interrupt status that was read in the interrupt service routine from the last interrupt serviced on the referenced channel.

NOTE: The TstVtx design is identical to the AtpVtx design except the I/O subsystem is replaced by frequency counters to verify the oscillator and PLL frequencies and the design ID is set to 0x55 rather than zero. The calls that differ are listed below:

IOCTL_TST_VTX_SET_BASE_CONFIG

Function: Writes the base control register configuration for the Virtex ATP design.
Input: ATP_VTX_BASE_CONFIG structure
Output: None
Notes: The CountClr and CountEn fields are not strictly useful, since the counters are automatically reset and enabled in the READ_COUNTER routine anyway. See the definition of TST_VTX_BASE_CONFIG below.

```
typedef struct _TST_VTX_BASE_CONFIG {
   BOOLEAN     Led0;
   BOOLEAN     Led1;
   BOOLEAN     Led2;
   BOOLEAN     Led3;
   BOOLEAN     CountEn;
   BOOLEAN     CountClr;
   BOOLEAN     ResetDcm;
} TST_VTX_BASE_CONFIG, *PTST_VTX_BASE_CONFIG;
```

IOCTL_TST_VTX_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: TST_VTX_BASE_CONFIG structure
Notes: Returns the values set in the preceding call.

IOCTL_TST_VTX_READ_COUNTER

Function: Returns the count from the 3-to-1 multiplexer of oscillator and PLL clock A and B.
Input: Counter to read (COUNT_SEL enumerated type)
Output: Counter value (unsigned long integer)
Notes: The counters are gated by the master counter, which is clocked by the reference oscillator. By examining the relative counts of the PLL clocked counters, the accuracy of the PLL clocks can be verified. Each time this is called, the counters are re-initialized and enabled, so subsequent counts of the same oscillator could differ slightly. See the definition of COUNT_SEL below.

```
typedef enum _COUNT_SEL {
   OSC,
   PLL_A,
   PLL_B
} COUNT_SEL, *PCOUNT_SEL;
```

IOCTL_TST_VTX_SET_CHAN_CONFIG

Function: Writes to a channel's control register.
Input: Channel number and configuration (TST_VTX_CHAN_CONFIG)
Output: None
Notes: There is no I/O system in this design, so the only function controlled by the channel control register is the transmitter to receiver FIFO loopback enable.

```
typedef struct _TST_VTX_CHAN_CONFIG {
   UCHAR        Channel;
   BOOLEAN      FifoTest;
} TST_VTX_CHAN_CONFIG, *PTST_VTX_CHAN_CONFIG;
```

IOCTL_TST_VTX_GET_CHAN_CONFIG

Function: Returns the configuration of the channel's control register.
Input: Channel number (unsigned character)
Output: TST_VTX_CHAN_CONFIG structure
Notes:

The IOCTLs defined for the GenVtx driver are described below:

IOCTL_GEN_VTX_GET_INFO

Function: Returns the Design ID, Driver version, Instance number and PLL device ID.
Input: None
Output: GEN_VTX_DRIVER_DEVICE_INFO structure
Notes: 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 GEN_VTX_DRIVER_DEVICE_INFO below.

```
typedef struct _GEN_VTX_DDINFO {
   UCHAR    DriverRev;
   UCHAR    PllDeviceId;
   UCHAR    DesignId;
   UCHAR    DesignRev;
   ULONG    InstanceNum;
} GEN_VTX_DDINFO, *PGEN_VTX_DDINFO;
```

IOCTL_GEN_VTX_SET_BASE_CONFIG

Function: Writes the base control register configuration for the Generic Virtex design.
Input: GEN_VTX_BASE_CONFIG structure
Output: None
Notes: See the definition of GEN_VTX_BASE_CONFIG below.

```
typedef struct _GEN_VTX_BASE_CONFIG {
   BOOLEAN     Led0;
   BOOLEAN     Led1;
   BOOLEAN     Led2;
   BOOLEAN     Led3;
   BOOLEAN     ResetDcm;
} GEN_VTX_BASE_CONFIG, *PGEN_VTX_BASE_CONFIG;
```

IOCTL_GEN_VTX_GET_BASE_CONFIG

Function: Returns the configuration of the base control register.
Input: None
Output: GEN_VTX_BASE_CONFIG structure
Notes: Returns the values set in the preceding call.

IOCTL_GEN_VTX_GET_BASE_STATUS

Function: Returns the value of the base status register.
Input: None
Output: Status value (unsigned long integer)
Notes: This call is used to determine if an interrupt is active. See the status bit definitions below.

```
#define GEN_VTX_STAT_DCM_LOCKED     0x00010000
#define GEN_VTX_STAT_INTSTAT0       0x00020000
#define GEN_VTX_STAT_INTSTAT1       0x00040000
```

IOCTL_GEN_VTX_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: GEN_VTX_PLL_DATA structure
Output: None
Notes: The GEN_VTX_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the data to write.

```
typedef struct _GEN_VTX_PLL_DATA {
   UCHAR    Data[PLL_MESSAGE_SIZE];
} GEN_VTX_PLL_DATA, *PGEN_VTX_PLL_DATA;
```

IOCTL_GEN_VTX_READ_PLL_DATA

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

IOCTL_GEN_VTX_WRITE_DATA

Function: Writes a data word to a specified address offset.
Input: GEN_VTX_WRITE_DATA
Output: None
Notes: The success of this call depends on the user's knowledge of the design's address map. No error checking is performed.

```
typedef struct _GEN_VTX_WRITE_DATA {
   UCHAR    AddrOffset;
   ULONG    Data;
} GEN_VTX_WRITE_DATA, *PGEN_VTX_WRITE_DATA;
```

IOCTL_GEN_VTX_READ_DATA

Function: Reads a data word from a specified address offset.
Input: Address offset (unsigned long integer)
Output: Data value (unsigned long integer)
Notes: The success of this call depends on the user's knowledge of the design's address map. No error checking is performed.

IOCTL_GEN_VTX_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_GEN_VTX_ENABLE_INTERRUPT

Function: Enables interrupts.
Input: None
Output: None
Notes: This command is run to allow the driver to respond to local interrupts. Interrupts are disabled in the driver interrupt service routine, therefore this command must be run after each interrupt occurs to re-enable interrupts.

IOCTL_GEN_VTX_DISABLE_INTERRUPT

Function: Disables interrupts.
Input: None
Output: None
Notes: This call is used when local interrupt processing is no longer desired.

IOCTL_GEN_VTX_FORCE_INTERRUPT

Function: Causes an interrupt to occur.
Input: None
Output: None
Notes: Causes an interrupt to be asserted on the PCI bus as long as interrupts are enabled. This IOCTL is used for development, to test interrupt processing.

IOCTL_GEN_VTX_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 for the last interrupt serviced.

Write

PMC-XM DMA data is written to one of the two XmChan devices 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 I/O.

Read

PMC-XM DMA data is read from one of the two XmChan devices 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 I/O.

Warranty and Repair

Dynamic Engineering warrants this product to be free from defects under normal use and service and in its original, unmodified condition, for a period of one year from the time of purchase. If the product is found to be defective within the terms of this warranty, Dynamic Engineering's sole responsibility shall be to repair, or at Dynamic Engineering's sole option to replace, the defective product.

Dynamic Engineering's warranty of and liability for defective products is limited to that set forth herein. Dynamic Engineering disclaims and excludes all other product warranties and product liability, expressed or implied, including but not limited to any implied warranties of merchantability or fitness for a particular purpose or use, liability for negligence in manufacture or shipment of product, liability for injury to persons or property, or for any incidental or consequential damages.

Dynamic Engineering's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Dynamic Engineering.

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing and in most cases it will be "cockpit error" rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call the Customer Service Department and arrange to speak with an engineer. We will work with you to determine the cause of the issue. If the issue is one of a defective driver we will correct the problem and provide an updated module(s) to you [no cost]. If the issue is of the customer's making [anything that is not the driver] the engineering time will be invoiced to the customer. Pre-approval may be required in some cases depending on the customer's invoicing policy.

Out of Warranty Repairs

Out of warranty support will be billed. The current minimum repair charge is $125. An open PO will be required.

For Service Contact:

Customer Service Department
Dynamic Engineering
150 DuBois, Suite C
Santa Cruz, CA 95060
(831)457-8891, Fax (831)457-4793
support@dyneng.com