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 Parallel TTL BA16 Base & Channel Software Manual

Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Manual Revision A

Corresponding Hardware: Revision A

10-2007-0101

BA16: Revision B

BA16Base & BA16Chan

WDM Device Drivers for the

PMC Parallel TTL BA16

Parallel TTL Interface w/ COS

Dynamic Engineering

150 DuBois, Suite C

Santa Cruz, CA 95060

(831) 457-8891

FAX: (831) 457-4793

Table of Contents

Introduction

The BA16Base and BA16Chan drivers were developed with the Windows Driver Foundation version 1.9 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The BA16 driver package has three parts. The driver is installed into the Windows® OS, the test executable and the User Application ñUserapò executable.

The driver and test are delivered as installed or executable items to be used directly or indirectly by the user. The Userapp code is delivered in source form [C] and is for the purpose of providing a reference to using the driver.

The ñtestò executable allows the user to use the driver in script form from a DOS window. Each driver call can be accessed, parameters set and returned. Normally not need or used by the integrator, but a very handy tool in certain circumstances. The test executable has a ñhelpò menu to explain the calls, parameters and returned information.

UserAp is a stand-alone code set with a simple and powerful menu plus a series of ñtestsò 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. For example most Dynamic Engineering PCI based designs support DMA. DMA is demonstrated with the memory based loop-back tests. The tests can be ported and modified to fit your requirements.

The test software can be ported to your application to provide a running start. It is recommended to port the switch and status tests to your application to get started. 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 hardware has features common to the board level and features that are set apart in ñchannelsò. The channels have the same offsets within the channel, and the same status and control bit locations allowing for symmetrical software in the calling routines. The driver supports the channels with a variable passed in to identify which channel is being accessed. The hardware manual defines the pinouts for each channel and the bitmaps and detailed configurations for each channel. The driver handles all aspects of interacting with the channels and base features.

We strive to make a useable product, and while we can guarantee operation we can't foresee all concepts for client implementation. If you have suggestions for extended features, special calls for particular set-ups or whatever please share them with us, [engineering@dyneng.com] and we will consider and in many cases add them.

The PMC Parallel TTL board has a Spartan3-1000 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for 64 IO. Each IO can be programmed to be an output or an input at any time. Each IO can have rising edge, falling edge or COS processing enabled. In addition the BA16 version has two transmit and two receive channels with byte wide DMA based IO. The driver supports programming a programmable PLL. Channel A of the PLL is used to control the transmit frequency on the channel based IO. Each channel has data FIFO's [2K Tx and 4K Rx].

When the PMC Parallel TTL BA16 board is recognized by the PCI bus configuration utility it will start the PmcParTtlBa16Base driver which will create a device object for each board, initialize the hardware, create child devices for the two I/O channels and request loading of the PmcParTtlBa16Chan driver. The PmcParTtlBa16Chan driver will create a device object for each of the I/O channels and perform initialization on each channel. 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 BA16 user manual (also referred to as the hardware manual).

Driver Installation

There are several files provided in each driver package. These files include Ba16BasePublic.h, Ba16Base.inf, ba16base.cat, Ba16Base.sys, Ba16ChanPublic.h, Ba16Chan.inf, ba16chan.cat, Ba16Chan.sys, and WdfCoInstaller01009.dll.

Ba16BasePublic.h and Ba16ChanPublic.h are the C header files that defines the Application Program Interface (API) for the driver. These file is required at compile time by any application that wishes to interface with the drivers, but is not needed for driver installation.

PmcTtlBa16UserApp.exe is a sample console applications that makes calls into the Ba16Base/Ba16Chan drivers to test each driver call without actually writing any application code. They are not required during driver installation either.

This test application is intended to test the proper functioning of each driver call, **not** for normal operation. Many integration efforts will never need the debugger capability that the test menu represents. The test capability will allow the designer to access the card without any other software in the way to make sure that the system can ñseeò the card and to do basic card manipulations.

Windows 7 Installation

Copy Ba16Base.inf, ba16base.cat, Ba16Base.sys, Ba16Chan.inf, ba16chan.cat, Ba16Chan.sys, and WdfCoInstaller01009.dll (Win7 version) to a CD or USB memory device as preferred.

With the 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 see the channels. Repeat this for each of the two channels as necessary.

***** 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 Ba16BasePublic.h and Ba16ChanPublic.h See main.c in the PmcTtlBa16UserApp 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 Ba16Base driver are described below:

IOCTL_BA16_BASE_GET_INFO

Function: Returns the device driver version, design ID, design rev, PLL device ID, user switch value, and device instance number.
Input: None
Output: BA16_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 BA16_BASE_DRIVER_DEVICE_INFO below. Bit definitions can be found under ópmcparttl_IDô section under Register Definitions in the Hardware manual.

```
// Driver/Device information
typedef struct _BA16_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverVersion;
   UCHAR    DesignID;
   UCHAR    DesignRev;
   UCHAR    PllDeviceId;
   UCHAR    SwitchValue;
   ULONG    InstanceNumber;
} BA16_BASE_DRIVER_DEVICE_INFO, *PBA16_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_BA16_BASE_LOAD_PLL

Function: Loads the internal registers of the PLL.
Input: BA16_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_BA16_BASE_READ_PLL

Function: Returns the contents of the PLLôs internal registers
Input: None
Output: BA16_BASE_PLL_DATA structure
Notes: The register data is output in the BA16_BASE_PLL_DATA structure In an array of 40 bytes

IOCTL_BA16_BASE_SET_DIR

Function: Write to Direction Register Lower (31-0) or Upper IO (63-32).

Input: BA16_BASE_ DIRECTION structure
Output: None
Notes: 0 = RX, and 1 = TX for each bit. See the definition of BA16_BASE_ DIRECTION below. Detailed defintions can be found under ópmcparttl_DirLôand ópmcparttl_DirUôsection under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_DIRECTION {
   ULONG   LowerDir;
   ULONG   UpperDir;
} BA16_BASE_DIRECTION, *PBA16_BASE_DIRECTION;
```

IOCTL_BA16_BASE_GET_DIR

Function: Read from Direction Register Lower (31-0) or Upper IO (63-32).
Input: None
Output: BA16_BASE_ DIRECTION structure
Notes: See the definition of BA16_BASE_DIRECTION above.

IOCTL_BA16_BASE_SET_DAT

Function: Write to Data Register Lower (31-0) or Upper IO (63-32).
Input: BA16_BASE_ DATA structure
Output: None
Notes: Bits written to register will go to IO if Parallel Enable bit is set and IO type is set to registered. See the definition of BA16_BASE_ DATA below. Detailed defintions can be found under ópmcparttl_DatLôand ópmcparttl_DatUôsection under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_DATA {
   ULONG   LowerData;
   ULONG   UpperData;
} BA16_BASE_DATA, *PBA16_BASE_DATA;
```

IOCTL_BA16_BASE_GET_DAT

Function: Read from Data IO Lower (31-0) or Upper IO (63-32).
Input: None
Output: BA16_BASE_ DATA structure
Notes: IO lines are read-back not register value ï may or may not match register. See the definition of BA16_BASE_ DATA above.

IOCTL_BA16_BASE_GET_DATREG

Function: Read from Data Register Lower (31-0) or Upper IO (63-32).
Input: None
Output: BA16_BASE_ DATAREG structure
Notes: SET_DAT Register data read-back. See the definition of BA16_BASE_ DATAREG below. Detailed defintions can be found under ǿpmcparttl_DatLregôand ǿpmcparttl_DatUregô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_DATAREG {
   ULONG   LowerDataReg;
   ULONG   UpperDataReg;
} BA16_BASE_DATAREG, *PBA16_BASE_DATAREG;
```

IOCTL_BA16_BASE_SET_PAREN

Function: Sets the parallel enable bit.
Input: None
Output: None
Notes*:* The Parallel Enable bit is used to enable the register IO to be clocked through to the external IO. If set the upper and lower IO will be updated when written. If cleared, the data registers updated and then set the IO will update coherently.

IOCTL_BA16_BASE_CLR_PAREN

Function: Clears the parallel enable bit.
Input: None
Output: None
Notes: Clearing the Parallel Enable will prevent changes to the data registers from changing the IO. Use to hold off updates for upper and lower IO to be synchronized.

IOCTL_BA16_BASE_SET_COSCLK

Function: Write to COS clock register.
Input: BA16_BASE_COS_CLOCK
Output: None
***Notes**:* Please note that the COS clock can be driven to an IO pin to verify frequency with a scope. Detailed bit defines can be found in the ópmcparttl_COSclkô section of the Register Definitions in the Hardware manual. The following is a quick summary to allow the basic function to be used without further research. Also see definition of BA16_BASE_COS_CLOCK below.

COS Clock definitions

11-0 = divisor, {reference / 2*(n+1)}, n>=1

PRE_PCI	0x0000 // select PCI clock for reference clock
PRE_OSC	0x2000 // select oscillator for reference clock
PRE_EXT	0x4000 // select external clock for reference clock
PRE_SPARE	0x6000 // spare set to pci clock
POST_SELECT_DIV	0x1000 // select divided clock
POST_SELECT_REF	0x0000 // select reference clock
COS_REF_D0_OUT	0x8000 // enable COS clock onto data 0, requires output direction set too

The clock for Change of State can be driven directly from the source or divided down from the source [POST_SELECT_] controls this option. To select the source choose PCI, Osc [50 MHz], External or Spare [set to the PCI clock currently].

If the divided version is desired use the Formula shown to program ñnò to get the frequency you need. For example with a 50 MHz [Osc] reference and N set to 24 the COS hardware will use a 1 MHz clock [50 MHz / 2 * (24+1)].

Allow sufficient time for the clock to stabilize prior to enabling COS operation.

```
typedef struct _BA16_BASE_COS_CLOCK {
   ULONG           Divisor;
   ULONG           PostSelect;
   COS_PRESELECT   PreSelect;
   ULONG           DatOut0En;
} BA16_BASE_COS_CLOCK, *PBA16_BASE_COS_CLOCK;
```

IOCTL_BA16_BASE_GET_COSCLK

Function: Read from COS clock register.
Input: None
Output: BA16_BASE_COS_CLOCK
Notes: Reading provides the current values in the COS Clock definition register with no side effects ï can read at any time without affecting the clock.

IOCTL_BA16_BASE_SET_RISEFALLREG

Function: Write to COS Rising or Falling Register bit enables Lower (31-0) or Upper IO (63-32).
Input: BA16_BASE_ RF_REG structure
Output: None
Notes: Select which bits are tested for rising and falling edge activity. See the definition of BA16_BASE_ RF_ REG below. Detailed defintions can be found under ópmcparttl_RisLregô, ópmcparttl_RisUregô, ópmcparttl_FallLregô, and ópmcparttl_FallUregô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_RF_REG {
   ULONG   LowerRisReg;
   ULONG   UpperRisReg;
   ULONG   LowerFallReg;
   ULONG   UpperFallReg;
} BA16_BASE_RF_REG, *PBA16_BASE_RF_REG;
```

IOCTL_BA16_BASE_GET_RISEFALLREG

Function: Read from COS Rising or Falling Register bit enables Lower (31-0) or Upper IO (63-32).
Input: None
Output: BA16_BASE_ RF_REG structure
Notes: No side effects from reading.

IOCTL_BA16_BASE_SET_INTREG

Function: Write to COS Interrupt Rising or Falling Register interrupt enables Lower (31-0) or Upper IO (63-32).
Input: BA16_BASE_ INT_REG structure
Output: None
Notes: Enable the interrupt corresponding to the rising COS status for each bit. Not setting the interrupt will allow polled operation using the status. See the definition of BA16_BASE_ INT_REG below. Detailed defintions can be found under ópmcparttl_IntRisLregô, ópmcparttl_IntRisUregô, ópmcparttl_IntFallLregô, and ópmcparttl_IntFallUregô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_INT_REG {
   ULONG   LowerIntRisReg;
   ULONG   UpperIntRisReg;
   ULONG   LowerIntFallReg;
   ULONG   UpperIntFallReg;
} BA16_BASE_INT_REG, *PBA16_BASE_INT_REG;
```

IOCTL_BA16_BASE_GET_INTREG

Function: Read from COS Rising or Falling Register bit enables Lower (31-0) or Upper IO (63-32).
Input: None
Output: BA16_BASE_ RF_REG structure
Notes: No side effects from reading.

IOCTL_BA16_BASE_CLR_INTSTAT

Function: Clears the interrupts for Rising or Falling Lower (31-0) or Upper (63-32) Register Interrupt Status.
Input: BA16_BASE_CLEAR_STAT
Output: None
Notes: Writes to the registers which clears the interrupts on a bit by bit basis. See the definition of BA16_BASE_ CLEAR_STAT below. Detailed defintions can be found under ópmcparttl_IntRisLrstatô, ópmcparttl_IntRisUstatô, ópmcparttl_IntFallLstatô, and ópmcparttl_IntFallUstatô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_CLEAR_STAT {
   ULONG    LowerRisStat;
   ULONG    UpperRisStat;
   ULONG    LowerFallStat;
   ULONG    UpperFallStat;
} BA16_BASE_CLEAR_STAT, *PBA16_BASE_CLEAR_STAT;
```

IOCTL_BA16_BASE_GET_INTSTAT

Function: Read from COS Interrupt Falling or Rising Lower (31-0) or Upper (63-32) Register Interrupt Status
Input: None
Output: BA16_BASE_INT_STAT
Notes: Read the status register to see which bits have been set indicating that a falling event has occurred for a programmed bit. It is recommended that the Interrupt status is cleared each time the enabled bits are changed. Writing back the data read will clear only the bits that the SW has registered as interrupts and will prevent missing interrupt events. See the definition of BA16_BASE_ INT_STAT below. Detailed defintions can be found under ópmcparttl_IntRisLrstatô, ópmcparttl_IntRisUstatô, ópmcparttl_IntFallLstatô, and ópmcparttl_IntFallUstatô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_INT_STAT {
   ULONG   LowerRisStat;
   ULONG   UpperRisStat;
   ULONG   LowerFallStat;
   ULONG   UpperFallStat;
} BA16_BASE_INT_STAT, *PBA16_BASE_INT_STAT;
```

IOCTL_BA16_BASE_SET_DR

Function: Write to DR Lower or Upper Register bit wise selection of DMA/State-machine control or register control of IO.
Input: BA16_BASE_ DRREG structure
Output: None
Notes: Select Register base or DMA / State-machine based IO. When ó0ô the register IO is selected. When ó1ô the BA16 function IO is selected. The BA16 function has inputs on the lower channels and outputs on the upper. The input function needs to be programmed as an input in the DIR registers. See the definition of BA16_BASE_ DRREG below. Detailed defintions can be found under ópmcparttl_DR_Lô and ópmcparttl_DR_Uô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_BASE_DRREG {
   ULONG   LowerDrSel;
   ULONG   UpperDrSel;
} BA16_BASE_DRREG, *PBA16_BASE_DRREG;
```

IOCTL_BA16_BASE_GET_DR

Function: Read from DR Lower or Upper Register.
Input: None
Output: BA16_BASE_ RF_REG structure
Notes: No side effects from reading.

IOCTL_BA16_BASE_GET_STATUS

Function: Returns Status Register
Input: None
Output: ULONG
Notes: For general purpose ï bit mapped access from the register.

IOCTL_BA16_BASE_SET_MASTEREN

Function: Sets master interrupt enable bit.
Input: None
Output: None
Notes: The Master Interrupt enable is needed to allow interrupts from some sources to be asserted. Please refer to the HW manual for details.

IOCTL_BA16_BASE_CLR_MASTEREN

Function: Clears master interrupt enable bit.
Input: None
Output: None
Notes: The Clear function can be used to disable some board level interrupt sources.

The IOCTLs defined for the Ba16Chan driver are described below:

IOCTL_BA16_CHAN_GET_INFO

Function: Returns the Instance Number and the Current Driver Version.
Input: None
Output: BA16_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of BA16_CHAN_DRIVER_DEVICE_INFO below.

```
typedef struct _BA16_CHAN_DRIVER_DEVICE_INFO {
   UCHAR     DriverVersion;
   ULONG     InstanceNumber;
} BA16_CHAN_DRIVER_DEVICE_INFO, *PBA16_CHAN_DRIVER_DEVICE_INFO;
```

IOCTL_BA16_CHAN_GET_STATUS

Function: Returns the value of the status register and clear latched bits.
Input: None
Output: Status register value (ULONG)
Notes: Latched interrupt status bits are cleared by read ï [call writes back and clears bits]. Detailed defintions can be found under ópmcparttl_ch0,1_stô section under Register Definitions in the Hardware manual.

IOCTL_BA16_CHAN_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: BA16_CHAN_FIFO_LEVELS structure
Output: None
Notes: The FIFO counts are compared to these levels to determine the value of the STAT_TX_FF_AMT and STAT_RX_FF_AFL status bits. See the definition of BA16_CHAN_FIFO_ LEVELS below. Detailed defintions can be found under ópmcparttl_ch0,1_tx_aecnt and ópmcparttl_ch0,1_rx_afcntô section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_CHAN_FIFO_LEVELS
{
   USHORT   AlmostFull;
   USHORT   AlmostEmpty;
} BA16_CHAN_FIFO_LEVELS, *PBA16_CHAN_FIFO_LEVELS;
```

IOCTL_BA16_CHAN_GET_FIFO_LEVELS

Function: Returns the transmitter almost empty and receiver almost full levels for the channel.
Input: None
Output: BA16_CHAN_FIFO_LEVELS structure
Notes: The FIFO counts are compared to these levels to determine the value of the STAT_TX_FF_AMT and STAT_RX_FF_AFL status bits.

IOCTL_BA16_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in transmit and receive FIFOs.
Input: None
Output: BA16_CHAN_FIFO_COUNTS structure
Notes: Returns the actual FIFO data counts. The Status register has a second Data count for the RX FIFO that includes the Pipeline between the FIFO and PCI bus. TX FIFO is 2K-1 deep, RX is 4K-1 deep.

IOCTL_BA16_CHAN_RESET_FIFOS

Function: Resets both FIFOs for the referenced channel.
Input: None
Output: None
Notes: Resets transmit and receive FIFOâs.

IOCTL_BA16_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.

IOCTL_BA16_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 interrupt occurs to re-enable it.

IOCTL_BA16_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_BA16_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. Board level master interrupt also needs to be set.

IOCTL_BA16_CHAN_GET_ISR_STATUS

Function: Returns the interrupt status read in the ISR from the last user interrupt.
Input: None
Output: Interrupt status value (ULONG)
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 interrupts that deal with the DMA transfers do not affect this value. Masked version of channel status.

IOCTL_BA16_CHAN_SWW_TX_FIFO

Function: Writes a 32-bit data word to the transmit FIFO.
Input: FIFO word (ULONG)
Output: none
Notes: Used to make single-word accesses to the transmit FIFO instead of using DMA.

IOCTL_BA16_CHAN_SWR_RX_FIFO

Function: Returns a 32-bit data word from the receive FIFO.
Input: None
Output: FIFO word (ULONG)
Notes: Used to make single-word accesses to the receive FIFO instead of using DMA.

IOCTL_BA16_CHAN_SET_CONT

Function: write to Channel Control register using structure
Input: BA16_CHAN_CONT
Output: None
Notes: See the definition of BA16_CHAN_CONT below. Detailed defintions can be found under "pmcparttl_ch0,1_base" section under Register Definitions in the Hardware manual.

```
typedef struct _BA16_CHAN_CONT {
   BOOLEAN            FifoTestEn;// BiPass Mode Control
   BOOLEAN            EnableTx;  // start transmit state machine or stop
   BA16_CHAN_MODE_SEL TXMODE;    // BA16_8, BA16_16, BA16_32, BA16_64 bit operation
- only 8 is legal on BA16
   BOOLEAN            TxEndian;  // Set for reversed byte pattern on transmit
   BOOLEAN            TxMtMode;  // T for pause mode, F to stop on empty FIFO
   BOOLEAN            TxClkSel;  // T for PLLA for reference, F for oscillator [50
mhz] set to False in BiPass mode
   BOOLEAN            EnableRx;  // start or stop RX state machine
   BOOLEAN            RxEndian;  // Set for reversed byte pattern on receive
   BOOLEAN            RxClkSel;  // T for PLLB for reference clock, F for osc [50
mhz].  Use 6x+ expected RX frequency
   BOOLEAN            WrDmaEn;   // Write DMA Interrupt Enable
   BOOLEAN            RdDmaEn;   // Read DMA Interrupt Enable
   BOOLEAN            RxIdle;    // Rx State Machine is Idle - read only
   BOOLEAN            TxIdle;    // Tx State Machine is Idle - read only
   BOOLEAN            ReadDmaIdle; // Read DMA SM is idle - read only
   BOOLEAN            WriteDmaIdle;// Write DMA SM is idle  - read only
} BA16_CHAN_CONT, *PBA16_CHAN_CONT;
```

IOCTL_BA16_CHAN_GET_CONT

Function: Read from Channel Control register using structure
Input: None
Output: BA16_CHAN_CONT structure
Notes:

IOCTL_BA16_CHAN_GET_RX_DAT_CNT

Function: Read from Channel Status register to get DataCount bit
Input: None
Output: RX data count (ULONG)
Notes:

Write

PMCTTLBA16 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

PMCTTLBA16 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 PMCTTLBA16 write and read are implemented with Kernel level write and read for high performance.

Please see ôfifo.cö in the PmcTtlBa16UserApp for examples.

Warranty

http://www.dyneng.com/warranty.html Please refer to the published warranty information.

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.

Development Support

Purchasers of Dynamic Engineering HW are provided with a small amount of integration support. Dynamic Engineering offers support packages to provide a block of time for software support. We can write additional calls, help debug client code, write additional user level situations to support your T&I.
Please see http://www.dyneng.com/TechnicalSupportFromDE.pdf for more details.

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
All information provided is Copyright Dynamic Engineering.