DYNAMIC ENGINEERING
150 DuBois St., 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

Driver Documentation

Win32 Driver Model

Manual Revision A
Corresponding Hardware: Revision A
10-2007-0101
Corresponding Firmware:
BA16: Revision B

BA16Base & BA16Chan
WDM Device Drivers for the
PMC Parallel TTL BA16
Parallel TTL Interface w/ COS

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

Manual Revision A Revised November 10, 2008

Table of Contents

Introduction

The BA16Base and BA16Chan drivers are Win32 driver model (WDM) device drivers for the PMC-Parallel-TTL BA16 from Dynamic Engineering.

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

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

The “test” 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 “help” 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 “tests” that can be run on the installed hardware. Each of the tests execute calls to the driver, pass parameters and structures, and get results back. With the sequence of calls demonstrated, the functions of the hardware are utilized for loop-back testing. The software is used for manufacturing test at Dynamic Engineering. 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 “channels”. 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 driver: BA16Base.sys, PMCBA16.inf, DDBA16Base.h, BA16BaseGUID.h, BA16Chan.sys, DDBA16Chan.h, BA16ChanGUID.h. Driver Test: BA16Test.exe, Userap: User Application source files.

BA16BaseGUID.h and BA16ChanGUID.h are C header files that define the device interface identifiers for the drivers. DDBA16Base.h and DDBA16Chan.h files 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.

BA16Test.exe is a sample Win32 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.

To run BA16Test, open a command prompt console window and type ***BA16Test -d0 -?*** to display a list of commands (the PmcParTtlBa16Test.exe file must be in the directory that the window is referencing). The commands are all of the form ***BA16Test -dn -im*** where ***n*** and ***m*** are the device number and PmcParTtlBa16Base driver ioctl number respectively or ***BA16Test -cn -im*** where ***n*** and ***m*** are the channel number (0-1) and PmcParTtlBa16Chan driver ioctl number respectively.

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 “see” the card and to do basic card manipulations.

Windows 2000 Installation

Copy PmcBA16.inf, BA16Base.sys and BA16Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Select ***Next***.
_ Select ***Search for a suitable driver for my device.***
_ Select ***Next***.
_ Insert the disk prepared above in the desired drive.
_ Select the appropriate drive e.g. ***Floppy disk drives***.
_ Select ***Next***.
_ The wizard should find the PmcBA16.inf file.
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the channels and reopen the ***New Hardware Wizard.*** Repeat this for each channel as necessary.

Windows XP Installation

Copy PmcBA16.inf, BA16Base.sys and BA16Chan.sys to a floppy disk, or CD if preferred. In some cases the files can be accessed over a network or from local HDD. Substitute the network address for the floppy instructions to proceed with an over the network installation.

With the hardware installed, power-on the PCI host computer and wait for the ***Found New Hardware Wizard*** dialogue window to appear.
_ Insert the disk prepared above in the desired drive.
_ Select ***No when asked to connect to Windows Update***.
_ Select ***Next***.
_ Select ***Install the software automatically.***
_ Select ***Next***.
_ Select ***Finish*** to close the ***Found New Hardware Wizard***.
The system should now see the channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

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 (GUIDs), which are defined in BA16BaseGUID.h and BA16ChanGUID.h.

The User Application software contains a file called “main.c”. Main has the initialization needed to get the handles to the base and channel assets of the installed PMC Parallel TTL BA16 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() (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 Ba16Base driver are described below:

IOCTL_BA16_BASE_GET_INFO
Function: Return the Instance Number, Switch value, PLL device ID, Xilinx rev and Current Driver Version
Input: None
Output: BA16_BASE_DRIVER_DEVICE_INFO : Structure
Notes: Switch value is the configuration of the on-board dip-switch that has been set by the User (see the board silk screen for bit position and polarity). The PLL ID is the device address of the PLL device. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDBA16Base.h for the definition of SPWR_BASE_DRIVER_DEVICE_INFO.

IOCTL_BA16_BASE_LOAD_PLL_DATA
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 analyzed 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_DATA
Function: Returns the contents of the PLL's 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_DIRL
Function: Write to Direction Register Lower IO 31-0
Input: ULONG
Output: none
Notes: 0 = Rx, 1 = Tx for each bit.

IOCTL_BA16_BASE_SET_DIRU
Function: Write to Direction Register Upper IO 63-32
Input: ULONG
Output: none
Notes: 0 = Rx, 1 = Tx for each bit.

IOCTL_BA16_BASE_GET_DIRL
Function: Read from Direction Register Lower IO 31-0
Input: none
Output: ULONG
Notes: 0 = Rx, 1 = Tx for each bit.

IOCTL_BA16_BASE_GET_DIRU
Function: Read From Direction Register Upper IO 63-32
Input: none
Output: ULONG
Notes: 0 = Rx, 1 = Tx for each bit.

IOCTL_BA16_BASE_SET_DATL
Function: Write to Data Register Lower bits 31-0
Input: ULONG
Output: none
Notes: Bits written to register will go to IO if Parallel Enable bit is set and IO type is set to registered [DRL and DATEN IOCTLs]

IOCTL_BA16_BASE_SET_DATU
Function: Write to Data Register Lower bits 63-32
Input: ULONG
Output: none
Notes: Bits written to register will go to IO if Parallel Enable bit is set and IO type is set to registered [DRL and DATEN IOCTLs]

IOCTL_BA16_BASE_GET_DATL
Function: Read from Data IO Lower bits 31-0
Input: none
Output: ULONG
Notes: IO lines are read-back not register value – may or may not match register

IOCTL_BA16_BASE_GET_DATU
Function: Read from Data IO Lower bits 63-32
Input: none
Output: ULONG
Notes: IO lines are read-back not register value – may or may not match register

IOCTL_BA16_BASE_GET_DATLREG
Function: Read from Data Register Lower bits 31-0
Input: none
Output: ULONG
Notes: SET DATL Register data read-back.

IOCTL_BA16_BASE_GET_DATUREG
Function: Read from Data Register Upper bits 63-32
Input: none
Output: ULONG
Notes: SET DATU Register data read-back.

IOCTL_BA16_BASE_SET_PAREN
Function: read base register then set parallel enable bit
Input: none
Output: ULONG Base Register value after enable is set.
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: read base register then clear parallel enable bit
Input: none
Output: ULONG Base Register value after enable is cleared.
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: short on ULONG boundary
Output: none
Notes: The bit defines are in DDBA16Base.h. Please note that the COS clock can be driven to an IO pin to verify frequency with a scope. Please see the HW manual for more information on the usage of the bits. The following is a quick summary to allow the basic functions to be used without further research.

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 “n” 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.

IOCTL_BA16_BASE_GET_COSCLK
Function: Read from COS clock register
Input: none
Output: short on ULONG boundary
Notes: reading provides the current values in the COS Clock definition register with no side affects – can read at any time without affecting the clock.

IOCTL_BA16_BASE_SET_RISLREG
Function: Write to COS Rising Lower Register bit enables 31-0
Input: ULONG
Output: none
Notes: Select which bits are tested for rising edge activity.

IOCTL_BA16_BASE_GET_RISLREG
Function: Read from COS Rising Lower Register bit enables 31-0
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_RISUREG
Function: Write to COS Rising Upper Register bit enables 63-32
Input: ULONG
Output: none
Notes: Select which bits are tested for rising edge activity.

IOCTL_BA16_BASE_GET_RISUREG
Function: Read from COS Rising Upper Register bit enables 63-32
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_FALLLREG
Function: Write to COS Falling Lower Register bit enables 31-0
Input: ULONG
Output: none
Notes: Select which bits are tested for Falling edge activity.

IOCTL_BA16_BASE_GET_FALLLREG
Function: Read from COS Falling Lower Register bit enables 31-0
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_FALLUREG
Function: Write to COS Falling Upper Register bit enables 63-32
Input: ULONG
Output: none
Notes: Select which bits are tested for Falling edge activity.

IOCTL_BA16_BASE_GET_FALLUREG
Function: Read from COS Falling Upper Register bit enables 63-32
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_INTRISLREG
Function: Write to COS Interrupt Rising Lower Register Interrupt Enables 31-0
Input: ULONG
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.

IOCTL_BA16_BASE_GET_INTRISLREG
Function: Read from COS Interrupt Rising Lower Register Interrupt Enables 31-0
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_INTRISUREG
Function: Write to COS Interrupt Rising Upper Register Interrupt Enables 63-32
Input: ULONG
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.

IOCTL_BA16_BASE_GET_INTRISUREG
Function: Read from COS Interrupt Rising Upper Register Interrupt Enables 63-32
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_INTFALLLREG
Function: Write to COS Interrupt Falling Lower Register Interrupt Enables 31-0
Input: ULONG
Output: none
Notes: Enable the interrupt corresponding to the Falling COS status for each bit. Not setting the interrupt will allow polled operation using the status.

IOCTL_BA16_BASE_GET_INTFALLLREG
Function: Read from COS Interrupt Falling Lower Register Interrupt Enables 31-0
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_SET_INTFALLUREG
Function: Write to COS Interrupt Falling Upper Register Interrupt Enables 63-32
Input: ULONG
Output: none
Notes: Enable the interrupt corresponding to the Falling COS status for each bit. Not setting the interrupt will allow polled operation using the status.

IOCTL_BA16_BASE_GET_INTFALLUREG
Function: Read from COS Interrupt Falling Upper Register Interrupt Enables 63-32
Input: none
Output: ULONG
Notes: no side affects from reading

IOCTL_BA16_BASE_CLR_INTRISLSTAT
Function: Write to COS Interrupt Rising Status Lower Register Interrupt Status
Input: ULONG
Output: none
Notes: Writing to the Interrupt Status register will clear the interrupts on a bit by bit basis.

IOCTL_BA16_BASE_GET_INTRISLSTAT
Function: Read from COS Interrupt Rising Status Lower Register Interrupt Status
Input: none
Output: ULONG
Notes: Read the status register to see which bits have been set indicating that a rising 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.

IOCTL_BA16_BASE_CLR_INTRISUSTAT
Function: Write to COS Interrupt Rising Status Upper Register Interrupt Status
Input: ULONG
Output: none
Notes: Writing to the Interrupt Status register will clear the interrupts on a bit by bit basis.

IOCTL_BA16_BASE_GET_INTRISUSTAT
Function: Read from COS Interrupt Rising Status Upper Register Interrupt Status
Input: none
Output: ULONG
Notes: Read the status register to see which bits have been set indicating that a rising 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.

IOCTL_BA16_BASE_CLR_INTFALLLSTAT
Function: Write to COS Interrupt Falling Status Lower Register Interrupt Status
Input: ULONG
Output: none
Notes: Writing to the Interrupt Status register will clear the interrupts on a bit by bit basis.

IOCTL_BA16_BASE_GET_INTFALLLSTAT
Function: Read from COS Interrupt Falling Status Lower Register Interrupt Status
Input: none
Output: ULONG
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.

IOCTL_BA16_BASE_CLR_INTFALLUSTAT
Function: Write to COS Interrupt Falling Status Upper Register Interrupt Status
Input: ULONG
Output: none
Notes: Writing to the Interrupt Status register will clear the interrupts on a bit by bit basis.

IOCTL_BA16_BASE_GET_INTFALLUSTAT
Function: Read from COS Interrupt Falling Status Upper Register Interrupt Status
Input: none
Output: ULONG
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.

IOCTL_BA16_BASE_SET_DRL
Function: Write to DR Lower Register bit wise selection of DMA/state-machine control or register control of IO
Input: ULONG
Output: none
Notes: Select Register base or DMA / State-machine based IO. When '0' the register IO is selected. When '1' 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.

IOCTL_BA16_BASE_GET_DRL
Function: Read from DR Lower Register
Input: ULONG
Output: none
Notes: No side affects from read

IOCTL_BA16_BASE_SET_DRU
Function: Write to DR Upper Register bit wise selection of DMA/state-machine control or register control of IO
Input: ULONG
Output: none
Notes: Select Register base or DMA / State-machine based IO. When '0' the register IO is selected. When '1' the BA16 function IO is selected. Bit by bit basis – all bits in the BA16 need to be selected for each channel that is to be used. Please note that the bits are not channelized in the DRL/DRU registers. The BA16 function has inputs on the lower channels and outputs on the upper. Please note that the output functions require the DIR bits to be set as well.

IOCTL_BA16_BASE_GET_DRU
Function: Read from DR Lower Register
Input: ULONG
Output: none
Notes: No side affects from read

Quick reference for DRU. Bit defines are in DDBA16Base.h

DR_U_CH0_ALL 0x000003FF // enable channel 0 data, align32, clockout
DR_U_CH1_ALL 0x03FF0000 // enable channel 1 data, align32, clockout

IOCTL_BA16_BASE_SET_BASEREG
Function: Write to Base Control Register - general access to base control register of card, use with bit definitions
Input: ULONG
Output: none
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_BA16_BASE_GET_BASEREG
Function: Read from Base Control Register - general access from base control register of card, use with bit definitions
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access to the base control register.

IOCTL_BA16_BASE_GET_STATUS
Function: Read from Status Register
Input: none
Output: ULONG
Notes: Use for general purpose – bit mapped access from the register. See DDBA1Base.h for bit map information. See the HW manual for exact definitions of bits.

IOCTL_BA16_BASE_SET_MASTEREN
Function: read base register then set master interrupt enable bit - read modify write to set master interrupt enable
Input: none
Output: updated Base Register contents
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: read base register then clear master interrupt enable bit - read mod write to clear master interrupt enable
Input: none
Output: updated Base Register contents
Notes: The Master Interrupt enable is needed to allow interrupts from some sources to Be asserted onto the bus. The Clr function can be used to disable some board level interrupt sources.

The IOCTLs defined for the PmcParTtlBa16Chan driver are described below:

IOCTL_BA16_CHAN_GET_INFO
Function: Return the Instance Number and Current Driver Version
Input: None
Output: BA16_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of BA16_CHAN_DRIVER_DEVICE_INFO in the DDBA16Chan.h header file.

IOCTL_BA16_CHAN_GET_STATUS
Function: Return 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 – [call writes back and clears bits]. See quick reference status bits below. Defines available in DDBA16Chan.h Detailed definitions are available in the HW manual.

STAT_TX_FF_MT	0x00000001
STAT_TX_FF_AMT	0x00000002
STAT_TX_FF_FL	0x00000004
STAT_TX_FF_VLD	0x00000008
STAT_RX_FF_MT	0x00000010
STAT_RX_FF_AFL	0x00000020
STAT_RX_FF_FL	0x00000040
STAT_RX_FF_VLD	0x00000080
STAT_TX_INT	0x00000100
STAT_RX_INT	0x00000200
STAT_TX_FF_INT	0x00000400
STAT_RX_FF_INT	0x00000800
STAT_WR_DMA_ERR	0x00001000
STAT_RD_DMA_ERR	0x00002000
STAT_WR_DMA_INT	0x00004000
STAT_RD_DMA_INT	0x00008000

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.

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:

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 one or both FIFOs for the referenced channel.
Input: BA16_FIFO_SEL enumeration type
Output: None
Notes: Resets transmit and receive FIFO's . Structure retained from other projects with independent reset capability.

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 (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 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 (unsigned long integer)
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 (unsigned long integer)
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 DDBA16Chan.h for structure. See below for quick reference.

IOCTL_BA16_CHAN_GET_CONT
Function: Read from Channel Control register using structure
Input: None
Output: BA16_CHAN_CONT
Notes: See DDBA16Chan.h for structure. See below for quick reference.

```
FifoTestEn;   // BiPass Mode Control
EnableTx;     // start transmit state machine or stop, can be auto cleared
TXMODE;       // BA16_8, BA16_16, BA16_32, BA16_64 bit operation
              - only 8 is legal on BA16
TxEndian;     // Set to use reversed byte pattern on transmit
TxMtMode;     // True to use pause mode, False to stop on empty [FIFO]
TxClkSel;     // True to use PLLA for reference, False to use oscillator [50 mhz]  set to
              False in BiPass mode
TxClkOutSel; // True to drive reference clock
EnableRx;     // start or stop RX state machine
RxEndian;     // Set to use reversed byte pattern on receive
RxClkSel;     // True to use PLLB for reference clock, False to use osc [50 mhz].  Use
              6x+ expected RX frequency
MIntEn;       // Master Interrupt Enable
WrDmaEn;      // Write DMA Interrupt Enable
RdDmaEn;      // Read DMA Interrupt Enable
RxIdle;       // Rx State Machine is Idle - read only
TxIdle;       // Tx State Machine is Idle - read only
ReadDmaIdle;   // Read DMA State Machine is idle - read only
WriteDmaIdle;   // Write DMA State Machine is idle - read only
```

Write

DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Win32 function WriteFile() and passing in the handle to the I/O channel 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

DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Win32 function ReadFile() and passing in the handle to the I/O channel device opened with CreateFile(), a pointer to a pre-allocated buffer that will contain the data read, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually read, and a pointer to an optional Overlapped structure for performing asynchronous IO.

Warranty and Repair

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 merchandisability 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 Street, Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax

support@dyneng.com