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

SpWrBase
&
SpWrChan

Driver Documentation

Developed with Windows Driver Foundation Ver1.9

Manual Revision B
Corresponding Hardware: 10-2004-0805/0808
PMC-SpaceWire Revision E
Corresponding Hardware: 10-2006-0102
PCI-SpaceWire Revision B
Corresponding Hardware: 10-2008-1101
ccPMC-SpaceWire Revision A
Corresponding Hardware: 10-2009-0902
PC104p-SpaceWire Revision B
Corresponding Firmware: Revision K

SpWrBase, SpWrChan
WDF Device Drivers for the
(cc)PMC/PCI/PC104p-SpaceWire
4-Channel SpaceWire Interface

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

Manual Revision B: Revised November 9, 2017

Table of Contents

Introduction

The SpWrBase and SpWrChan drivers are Windows device drivers for the PMC-SpaceWire, ccPMC-SpaceWire, PCI-SpaceWire or PC104p-SpaceWire 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).

These SpaceWire boards have either a Spartan3-1500 Xilinx FPGA, Spartan3-2000 Xilinx FPGA, or a Spartan6-100 Xilinx FPGA to implement the PCI interface, internal FIFOs and protocol control and status for four SpaceWire channels. There is also a programmable PLL with four clock outputs to create separate programmable I/O clocks for each SpaceWire channel. Each channel has two 1023 x 32-bit packet-length FIFOs and either two 1k x 32-bit internal data FIFOs; or a 1k x 32-bit internal transmit data FIFO and a 128k x 32-bit external receive data FIFO; or two 128k x 32-bit external data FIFOs.

The SpWrChan driver will recognize and control an I/O channel using any of these internal/external FIFO combinations. When the channel driver initializes, it checks each channel for internal or external FIFOs. Default values for the transmit FIFO almost empty and receive FIFO almost full values are written to the appropriate registers (⅛ and ⅞ of the FIFO size respectively). TX and RX FIFO sizes are reported in the channel info structure.

When the SpaceWire board is recognized by the PCI bus configuration utility it will load the SpWrBase driver which will create a device object for each board, initialize the hardware, create child devices for the four I/O channels and request loading of the SpWrChan driver. The SpWrChan 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 I/O channel devices.

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 SpaceWire hardware manual.

Driver Installation

There are several files provided in each driver package. These files include SpWrPublic.h, SpWrBase.inf, SpWrBase.cat, SpWrBase.sys, SpWrBasePublic.h, SpWrChan.inf, SpWrChan.cat, SpWrChan.sys, SpWrChanPublic.h, WdfCoInstaller01009.dll.

SpWrBasePublic.h and SpWrChanPublic.h are C header files that define the Application Program Interface (API) for the SpWrBase and SpWrChan drivers. These files are required at compile time by any application that wishes to interface with the drivers, but are not needed for driver installation.

Windows XP Installation

Copy SpWrBase.inf, SpWrBase.cat, SpWrBase.sys, SpWrChan.inf, SpWrChan.cat, SpWrChan.sys and WdfCoInstaller01009.dll (XP version) to a floppy disk, CD or USB memory device or other system accessible location as preferred.

With the PMC-SpaceWire 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 SpWr I/O channels and reopen the ***New Hardware Wizard***. Proceed as above for each channel as necessary.

Windows 7 Installation

Copy SpWrBase.inf, SpWrBase.cat, SpWrBase.sys, SpWrChan.inf, SpWrChan.cat, SpWrChan.sys and WdfCoInstaller01009.dll (Win7 version) to a floppy disk, CD or USB memory device or other system accessible location as preferred.

With the PMC-SpaceWire 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 SpWr I/O channels in the Device Manager.
- Right-click on each channel icon, select ***Update Driver Software*** and proceed as above for each channel 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 SpWrBasePublic.h and SpWrChanPublic.h. See main.c in the SpWrUserApp project for an example of how to acquire handles for the base and four channel devices.

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

IO Controls

The drivers use IO Control calls (IOCTLs) to configure the device. IOCTLs refer to a single Device Object, which controls a single board or I/O channel. IOCTLs are called using the Win32 function DeviceIoControl() (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 SpWrBase driver are described below:

IOCTL_SPWR_BASE_GET_INFO

Function: Returns the device driver version, design version, design type, user switch value, device instance number and PLL device ID.
Input: None
Output: SPWR_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 SPWR_BASE_DRIVER_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _SPWR_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignRev;
   UCHAR    DesignType;
   UCHAR    SwitchValue;
   ULONG    InstanceNum;
   UCHAR    PllDeviceId;
} SPWR_BASE_DRIVER_DEVICE_INFO, *PSPWR_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_SPWR_BASE_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input:
SPWR_BASE_PLL_DATA structure
Output: None
Notes: The SPWR_BASE_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the PLL register data to write. See below for the definition of SPWR_BASE_PLL_DATA.

```
#define PLL_MESSAGE1_SIZE  16
#define PLL_MESSAGE2_SIZE  24
#define PLL_MESSAGE_SIZE  (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _SPWR_BASE_PLL_DATA {
   UCHAR  Data[PLL_MESSAGE_SIZE];
} SPWR_BASE_PLL_DATA;
```

IOCTL_SPWR_BASE_READ_PLL_DATA

Function: Returns the contents of the internal registers of the PLL.
Input: None
Output: SPWR_BASE_PLL_DATA structure
Notes: The register data is written to the SPWR_BASE_PLL_DATA structure in an array of 40 bytes. See definition of SPWR_BASE_PLL_DATA above.

IOCTL_SPWR_BASE_SET_TIME_CONFIG

Function: Sets the time-code timing and routing on the SpaceWire board.
Input: SPWR_BASE_TIME_CONFIG structure
Output: None
Notes: The master counter that controls the TICK_IN rate is clocked by the 80 MHz link clock. Count, in the input data structure is the count at which the master counter will roll-over, increment the six-bit time-code count and issue a TICK_IN pulse. Flags specifies the two control flag bits sent in bit 6 and 7 of the time-code data byte. TimeSource is a four-value array of SPWR_TM_SRC values that determine the source of time-codes sent by each of the four channels. These values specify one of the following six time-code sources: Master timer, any of the four channel's time-code outputs, or none (disabled). See below for the definition of SPWR_TM_SRC SPWR_BASE_TIME_CONFIG.

```
typedef enum _SPWR_TM_SRC {
   SPWRDISABLE,
   SPWRMASTER,
   SPWRCHAN0,
   SPWRCHAN1,
   SPWRCHAN2,
   SPWRCHAN3
} SPWR_TM_SRC, *PSPWR_TM_SRC;

typedef struct _SPWR_BASE_TIME_CONFIG {
   ULONG          Count;
   UCHAR          Flags;
   SPWR_TM_SRC    TimeSource[SPWR_NUM_CHANNELS];
} SPWR_BASE_TIME_CONFIG, *PSPWR_BASE_TIME_CONFIG;
```

IOCTL_SPWR_BASE_GET_TIME_CONFIG

Function: Returns the time-code timing and routing on the SpaceWire board.
Input: None
Output: SPWR_BASE_TIME_CONFIG structure
Notes: Returns the values set in the previous call.

The IOCTLs defined for the SpWrChan driver are described below:

IOCTL_SPWR_CHAN_GET_INFO

Function: Returns the driver version, instance number and transmit and receive FIFO sizes.
Input: None
Output: SPWR_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of SPWR_CHAN_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _SPWR_CHAN_DRIVER_DEVICE_INFO {
   UCHAR             DriverRev;
   UCHAR             ChannelNum;
   ULONG             InstanceNum;   // Board instance from base driver
   UCHAR             SwitchValue;   // Board user switch value from base
   UCHAR             DesignRev;     // Design revision from base driver
   UCHAR             DesignType;    // Design type from base driver
   ULONG             TxFifoSize;
   ULONG             RxFifoSize;
   SPWR_PKTLEN_TYPE  PktLenType;
} SPWR_CHAN_DRIVER_DEVICE_INFO, *PSPWR_CHAN_DRIVER_DEVICE_INFO;

typedef enum _SPWR_PKTLEN_TYPE {
   SPWR_16K,    // Original packet-length  16 KByte max
   SPWR_128K,   // Upgraded packet-length 128 KByte max
   SPWR_2G      // Upgraded packet-length   2 GByte max
} SPWR_PKTLEN_TYPE;
```

IOCTL_SPWR_CHAN_SET_CONFIG

Function: Specifies the channel control configuration.
Input: SPWR_CHAN_CONFIG structure
Output: None
Notes: Specifies the link startup behavior, enabled interrupt sources, DMA preemption behavior, DMA status and other control parameters. See the definitions of SPWR_START, SPWR_INTS, SPWR_DMA_PRMPT, SPWR_DMA_STAT and SPWR_CHAN_CONFIG below.

```
typedef enum _SPWR_START {
   SPWR_STOP,                // Channel link not connected
   SPWR_ISTRT,               // Channel initiates link
   SPWR_ASTRT                // Channel waits for a NULL to be received
} SPWR_START, *PSPWR_START;

typedef struct _SPWR_INTS {
   BOOLEAN  TxAmtInt;        // Transmit FIFO almost empty interrupt
   BOOLEAN  RxAflInt;        // Receive FIFO almost full interrupt
   BOOLEAN  RxErrInt;        // Reception error interrupt
   BOOLEAN  RxPktInt;        // Packet received interrupt
   BOOLEAN  TmTckInt;        // Time-code tick interrupt
} SPWR_INTS, *PSPWR_INTS;
 // Channel DMA priority (use sparingly)
typedef enum _SPWR_DMA_PRMPT {
   SPWR_NONE,                // No priority
   SPWR_READ,                // Read DMA has priority
   SPWR_WRITE,               // Write DMA has priority
   SPWR_RDWR              // Read and Write DMA have priority
} SPWR_DMA_PRMPT, *PSPWR_DMA_PRMPT;

typedef enum _SPWR_DMA_STAT {
   SPWR_BUSY,             // Read and Write DMA both active
   SPWR_RD_RDY,           // Read DMA idle
   SPWR_WR_RDY,           // Write DMA idle
   SPWR_BOTH_RDY          // Read and Write DMA both idle
} SPWR_DMA_STAT, *PSPWR_DMA_STAT;

typedef struct _SPWR_CHAN_CONFIG {
   UCHAR          ClockDivide;  // (1..16)PLL frequency/ClockDivide=I/O bps
   SPWR_START     StartMode;    // Link start mode (initiate or auto-start)
   SPWR_INTS      IntConfig;    // Interrupt condition enables
   BOOLEAN        NoPackets;    // Disable packets
   BOOLEAN        ReusePktLen;  // Reuse a single packet-length
   BOOLEAN        VldRxPktLen;  // Return only valid Rx packet-lengths
   BOOLEAN        FifoBypassEn; // Enables auto tx->rx FIFO transfer
   SPWR_DMA_PRMPT DmaPriority;  // DMA preemption control
   SPWR_DMA_STAT  DmaStatus;    // DMA status (read-only)
} SPWR_CHAN_CONFIG, *PSPWR_CHAN_CONFIG;
```

IOCTL_SPWR_CHAN_GET_CONFIG

Function: Returns the fields set in the previous call.
Input: None
Output: SPWR_CHAN_CONFIG structure
Notes: See the definitions of SPWR_START, SPWR_INTS, SPWR_DMA_PRMPT, SPWR_DMA_STAT and SPWR_CHAN_CONFIG above.

IOCTL_SPWR_CHAN_GET_STATUS

Function: Returns the channel's status register value and clears the latched status bits.
Input: None
Output: Value of the channel's status register (unsigned long integer)
Notes: See the status bit definitions below. Only the bits in CHAN_STAT_MASK will be returned. The bits in CHAN_STAT_LATCH_MASK will be cleared by this call only if they are set when the register was read. This prevents the possibility of missing an interrupt condition that occurs after the register has been read but before the latched register bits are cleared.

```
 // Status bit definitions
#define CHAN_STAT_TX_FF_MT     0x00000001 // Transmit FIFO empty
#define CHAN_STAT_TX_FF_AMT    0x00000002 // Transmit FIFO almost empty
#define CHAN_STAT_TX_FF_FL     0x00000004 // Transmit FIFO full
#define CHAN_STAT_TX_FF_VLD    0x00000008 // Transmit data valid (data in waiting to send latch)
#define CHAN_STAT_RX_FF_MT     0x00000010 // Receive FIFO empty
#define CHAN_STAT_RX_FF_AFL    0x00000020 // Receive FIFO almost full
#define CHAN_STAT_RX_FF_FL     0x00000040 // Receive FIFO full
#define CHAN_STAT_RX_FF_VLD    0x00000080 // Receive data valid (data in pipeline (4 words))
#define CHAN_STAT_PAR_ERR      0x00000100 // Parity error
#define CHAN_STAT_DSCNCT       0x00000200 // Disconnect error (no activity for > 850 ns)
#define CHAN_STAT_ESC_ERR      0x00000400 // Escape error (invalid escape sequence)
#define CHAN_STAT_CRDT_ERR     0x00000800 // Credit error (transmit or receive credit violation)
#define CHAN_STAT_RX_OVFL      0x00001000 // Receive FIFO overflow (write attempt to full FIFO)
#define CHAN_STAT_RX_ERROR     0x00002000 // Receive error (combines above 5 errors)
#define CHAN_STAT_PKT_DONE     0x00004000 // Receive packet complete (EOP or EEP received)
#define CHAN_STAT_TICK_RCVD    0x00008000 // Tick-out received (valid timecode)
#define CHAN_STAT_WR_DMA_INT   0x00010000 // Write DMA interrupt (shown for info only)
#define CHAN_STAT_RD_DMA_INT   0x00020000 // Read DMA interrupt (shown for info only)
#define CHAN_STAT_WR_DMA_ERR   0x00040000 // Write DMA error (abort or descriptor error)
#define CHAN_STAT_RD_DMA_ERR   0x00080000 // Read DMA error (abort or descriptor error)
#define CHAN_STAT_LINKED       0x00100000 // True if channel is successfully linked
#define CHAN_STAT_TX_PURGERR   0x00200000 // Transmitter purge error
#define CHAN_STAT_RX_PKTVLD    0x00400000 // Receive packet-length available to read
#define CHAN_STAT_INT_ACTIVE   0x00800000 // Enabled interrupt condition is active
#define CHAN_STAT_TM_DATA_MASK 0x3F000000 // Timecode data word (six bits)
#define CHAN_STAT_TX_AMT_LT    0x40000000 // Transmit FIFO almost empty (latched)
#define CHAN_STAT_RX_AFL_LT    0x80000000 // Receive FIFO almost full (latched)

#define CHAN_STAT_FIFO_MASK   (CHAN_STAT_TX_FF_MT  | CHAN_STAT_TX_FF_AMT | CHAN_STAT_TX_FF_FL  |\
                               CHAN_STAT_TX_FF_VLD | CHAN_STAT_RX_FF_MT  | CHAN_STAT_RX_FF_AFL |\
                               CHAN_STAT_RX_FF_FL  | CHAN_STAT_RX_FF_VLD)

#define CHAN_STAT_LATCH_MASK  (CHAN_STAT_PAR_ERR | CHAN_STAT_WR_DMA_ERR | CHAN_STAT_CRDT_ERR |\
                               CHAN_STAT_DSCNCT  | CHAN_STAT_RD_DMA_ERR | CHAN_STAT_RX_ERROR |\
                               CHAN_STAT_ESC_ERR | CHAN_STAT_TX_AMT_LT  | CHAN_STAT_PKT_DONE |\
                               CHAN_STAT_RX_OVFL | CHAN_STAT_RX_AFL_LT  | CHAN_STAT_TX_PURGERR)

#define CHAN_STAT_MASK      (CHAN_STAT_WR_DMA_INT | CHAN_STAT_RX_PKTVLD | CHAN_STAT_LINKED|\
                             CHAN_STAT_RD_DMA_INT | CHAN_STAT_FIFO_MASK | CHAN_STAT_LATCH_MASK |\
                             CHAN_STAT_INT_ACTIVE | CHAN_STAT_TICK_RCVD | CHAN_STAT_TM_DATA_MASK)
```

IOCTL_SPWR_CHAN_WRITE_PACKET_LENGTH

Function: Writes a transmitter packet-length value to the packet-length FIFO.
Input: Packet length value (unsigned long integer)
Output: None
Notes: When operating in packet mode, no data will be sent until at least one value is written to the transmit packet-length FIFO. Setting bit 31 high causes the transmitted packet to be terminated with an EEP rather than an EOP.

IOCTL_SPWR_CHAN_READ_PACKET_LENGTH

Function: Reads a received packet-length value from the packet-length FIFO.
Input: None
Output: Packet length value (unsigned long integer)
Notes: Bits 30-0 are used for the packet-length (maximum of 2 G Bytes). Bit 31 is an error flag that indicates that an error condition occurred during the reception of the referenced packet or that it was terminated by an EEP. Reading the channel status will indicate whether a connection error was detected.

IOCTL_SPWR_CHAN_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: SPWR_CHAN_FIFO_LEVELS structure
Output: None
Notes: These values are initialized to the default values ⅛ FIFO and ⅞ FIFO respectively when the driver initializes. The FIFO counts are compared to these levels to set the value of the CHAN_STAT_TX_FF_AMT and CHAN_STAT_RX_FF_AFL status bits and latch the CHAN_STAT_TX_AMT_LT and CHAN_STAT_RX_AFL_LT latched status bits. Also if the control bits CHAN_CNTRL_URGNT_OUT_EN and/or CHAN_CNTRL_URGNT_IN_EN are set, the FIFO level values are used to determine when to give priority to an output or input DMA channel that is running out of data or room to store data. See the definition of SPWR_CHAN_FIFO_LEVELS below.

```
typedef struct _SPWR_CHAN_FIFO_LEVELS {
   ULONG    AlmostFull;
   ULONG    AlmostEmpty;
} SPWR_CHAN_FIFO_LEVELS, *PSPWR_CHAN_FIFO_LEVELS;
```

IOCTL_SPWR_CHAN_GET_FIFO_LEVELS

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

IOCTL_SPWR_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive data and packet-length FIFOs.
Input: None
Output: SPWR_CHAN_FIFO_COUNTS structure
Notes: There is one pipe-line latch for the transmit FIFO data and four for the receive FIFO data. These are counted in the FIFO counts. That means the transmit count can be a maximum of 1025 32-bit words and the receive count can be a maximum of 1028 32-bit words. The TxPktCount and RxPktCount fields can be a maximum of 1023 packet lengths. See the definition of SPWR_CHAN_FIFO_COUNTS below.

```
typedef struct _SPWR_CHAN_FIFO_COUNTS {
 ULONG  TxCount;   // Number of data words in the transmit data FIFO
 ULONG  RxCount;   // Number of data words in the receive data FIFO
 USHORT TxPktCount;// Number of values currently in the tx packet-length FIFO
 USHORT RxPktCount;// Number of values currently in the rx packet-length FIFO
} SPWR_CHAN_FIFO_COUNTS, *PSPWR_CHAN_FIFO_COUNTS;
```

IOCTL_SPWR_CHAN_RESET_FIFOS

Function: Resets one or both FIFOs for the referenced channel.
Input: SPWR_FIFO_SEL enumeration type
Output: None
Notes: Resets the transmit or receive FIFO or both depending on the input parameter selection. Also resets the corresponding packet-length FIFO(s) and sets the programmable almost full/empty levels back to the default values for the FIFO(s) that were reset. See the definition of SPWR_FIFO_SEL below.

```
 // Used for FIFO reset call
typedef enum _SPWR_FIFO_SEL {
   SPWR_TX,
   SPWR_RX,
   SPWR_BOTH
} SPWR_FIFO_SEL, *PSPWR_FIFO_SEL;
```

IOCTL_SPWR_CHAN_WRITE_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_SPWR_CHAN_READ_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_SPWR_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 this event to be signaled.

IOCTL_SPWR_CHAN_ENABLE_INTERRUPT

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

IOCTL_SPWR_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_SPWR_CHAN_FORCE_INTERRUPT

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

IOCTL_SPWR_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. If the TICK_OUT interrupt is not enabled, the time-code data will not be read in the ISR and the tick received latched status bit will not be cleared. The new field is true if the Status has been updated since it was last read. See the definition of SPWR_CHAN_INT_STAT below.

```
typedef struct _SPWR_CHAN_INT_STAT {
   ULONG     Status;
   BOOLEAN   New;
} SPWR_CHAN_INT_STAT, *PSPWR_CHAN_INT_STAT;
```

IOCTL_SPWR_CHAN_READ_TIME_CODE

Function: Returns the last time-code received and clears the tick received latched bit.
Input: None
Output: SPWR_CHAN_TIME_CODE structure
Notes: Returns the value of the time-code data byte last received in the Time field. The New field will be set to TRUE if the time-code has not been previously read. Either by a previous instance of this call or by an ISR responding to an enabled TICK_OUT interrupt. If the TICK_OUT interrupt is not enabled, the time-code data will not be read in the ISR and the tick received latched status bit will not be cleared. See the definition of SPWR_CHAN_TIME_CODE structure below.

```
typedef struct _SPWR_CHAN_TIME_CODE {
   UCHAR     Time;
   UCHAR     Flags;
   BOOLEAN   New;
} SPWR_CHAN_TIME_CODE, *PSPWR_CHAN_TIME_CODE;
```

IOCTL_SPWR_CHAN_READ_DMA_COUNTS

Function: Returns the word counts of the last input and output DMA transfers.
Input: None
Output: SPWR_CHAN_DMA_COUNTS structure
Notes: The number of words transferred is reported even if the transfer was cancelled before it completed. The word counters are only cleared when a new DMA command is initiated or when the channel is initialized.

```
typedef struct _SPWR_CHAN_DMA_COUNTS {
   ULONG     WriteCount;
   ULONG     ReadCount;
} SPWR_CHAN_DMA_COUNTS, *PSPWR_CHAN_DMA_COUNTS;
```

Write

SpaceWire 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

SpaceWire 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

Please refer to the warranty page on our website for the current warranty offered and options.

http://www.dyneng.com/warranty.html

Service Policy

The software described in this manual is provided at no cost to clients who have purchased the corresponding hardware. Minimal support is included along with the documentation. For help with integration into your project please contact sales@dyneng.com for a support contract. Several options are available. With a contract in place Dynamic Engineers can help with system debugging, special software development, or whatever you need to get going.

For Service Contact:

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