DYNAMIC ENGINEERING
150 DuBois, Suite C
Santa Cruz, CA 95060
(831) 457-8891
https://www.dyneng.com
sales@dyneng.com
Est. 1988

SpWrBkBase
&
SpWrBkChan

Win10 Driver Documentation

PMC-SpaceWire-BK
PCI-SpaceWire-BK
PCIe-SpaceWire-BK
PC104p-SpaceWire-BK

Developed with Windows Driver Foundation

Base Revision 01p2
Corresponding Hardware:
(PMC) 10-2004-0809,10,11
(PCI) 10-2006-0104, 05
(PCIe) 10-2018-1801, 02, 03
(PC104p) 10-2008-0903, 04

SpWrBkBase, SpWrBkChan
WDF Device Drivers for the
PMC/PCI/PCIe-SpaceWire-BK
4-Channel SpaceWireBk Interface

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

Revised 8/14/2019

Table of Contents

Introduction

The SpWrBkBase and SpWrBkChan drivers are Windows device drivers for PMC/PCI/PC104p/PCIe-SpaceWire-BK. These drivers were developed with the Windows Driver Foundation version 1.19 (WDF) from Microsoft, specifically the Kernel-Mode Driver Framework (KMDF).

The SpaceWire-BK design utilizes a Xilinx Spartan 6 FPGA to implement the PCI interface, FIFOs, protocol control and status for four SpaceWire-BK channels. A programmable PLL with four clock outputs creates a separate programmable I/O clock for each SpaceWire-BK port. Each channel has two 16k x 32-bit internal data FIFOs and two 1023 x 32-bit packet-length FIFOs. Select channels may have additional external 128k x 32-bit FIFOs. The -128 version has external FIFOs added to the TX and RX circuit of channel zero for a total of 144K x 32-bits. The -128RX version has external FIFOs added to the RX circuit of channels zero and one. These different versions are distinguished by the 4-bit Xilinx type field of the user info register. A one in this field indicates that all channels have only internal FIFOs, a two indicates the -128 version and a three indicates the -128RX version.

When the SpaceWire-BK board is recognized by the PCI bus configuration utility it will load the SpWrBkBase 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 SpWrBkChan driver. The SpWrBkChan 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 using 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 SpaceWire-BK hardware manual. In addition, the UserAp reference SW package has examples of initializing, configuring, and using the SpaceWire driver. The UserAp software is provided in source form to allow user adaptation into their system.

The driver package is compiled and signed for x64 standard systems [not ARM etc.].

Driver Installation

There are several files provided in each driver package. These files include SpWrBkPublic.h, SpWrBkBase.inf, SpWrBkBase.cat, SpWrBkBase.sys, SpWrBkBasePublic.h, SpWrBkChan.inf, SpWrBkChan.cat, SpWrBkChan.sys and SpWrBkChanPublic.h.

SpWrBkPublic.h, SpWrBkBasePublic.h and SpWrBkChanPublic.h are C header files that define the Application Program Interface (API) for the SpWrBkBase and SpWrBkChan drivers. These files are required at compile time by any application that wishes to interface with the drivers, and are not needed for driver installation.

Windows 10 Installation

Copy SpWrBkBase.inf, SpWrBkBase.cat, SpWrBkBase.sys, SpWrBkChan.inf, SpWrBkChan.cat and SpWrBkChan.sys to a USB drive or other removable memory device as preferred.

With the SpaceWire-BK hardware installed, power-on the host computer and open the Device Manager.

If ***Other PCI Bridge Device*** is not seen, select ***Scan for hardware changes*** from the ***Action*** menu.

Right-click on the Other PCI Bridge Device and select Update driver from the pop-up menu. Navigate to the memory device prepared above and select the folder containing the driver files.

Once the Base driver has been installed, the four Channel Devices will be seen in the Device Manager display. Right-click on each of the Channel Devices and select Update driver from the pop-up menu and proceed as before to install the Channel driver to the four Channel Devices.

Driver Startup

Once the drivers have been installed they 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 a globally unique identifier (GUID), which is defined in SpWrBkBasePublic.h and SpWrBkChanPublic.h. See main.c in the SpWrBkUserApp 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. See G_ALL.h within the UserAp file set.

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 SpWrBkBase driver are described below:

IOCTL_SPWRBK_BASE_GET_INFO

Function: Returns the device driver revision, design revision, design type, user switch value, device instance number and PLL device ID.
Input: None
Output: SPWRBK_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 SPWRBK_BASE_DRIVER_DEVICE_INFO below.

```
 // Driver/Device information
typedef struct _SPWRBK_BASE_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    DesignRev;      // Design revision
   UCHAR    DesignRevMin;   // Design minor revision
   UCHAR    DesignType;     // Design type
   ULONG    InstanceNum;    // Board instance
   UCHAR    SwitchValue;    // Board user switch value
   UCHAR    PllDeviceId;
   BOOLEAN  BridgeCnfgd;
 } SPWRBK_BASE_DRIVER_DEVICE_INFO, *PSPWRBK_BASE_DRIVER_DEVICE_INFO;
```

IOCTL_SPWRBK_BASE_LOAD_PLL_DATA

Function: Writes to the internal registers of the PLL.
Input:
SPWRBK_BASE_PLL_DATA structure
Output: None
Notes: The SPWRBK_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 SPWRBK_BASE_PLL_DATA.

```
 // Structures for IOCTLs
#define PLL_MESSAGE1_SIZE  16
#define PLL_MESSAGE2_SIZE  24
#define PLL_MESSAGE_SIZE  (PLL_MESSAGE1_SIZE + PLL_MESSAGE2_SIZE)

typedef struct _SPWRBK_BASE_PLL_DATA {
   UCHAR  Data[PLL_MESSAGE_SIZE];
} SPWRBK_BASE_PLL_DATA;
```

IOCTL_SPWRBK_BASE_READ_PLL_DATA

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

IOCTL_SPWRBK_BASE_SET_TIME_CONFIG

Function: Sets the time-code timing and routing on the SpaceWire-BK board.
Input: SPWRBK_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 SPWRBK_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 SPWRBK_TM_SRC SPWRBK_BASE_TIME_CONFIG.

```
typedef enum _SPWRBK_TM_SRC {
   SPWRBKDISABLE,
   SPWRBKMASTER,
   SPWRBKCHAN0,
   SPWRBKCHAN1,
   SPWRBKCHAN2,
   SPWRBKCHAN3
} SPWRBK_TM_SRC, *PSPWRBK_TM_SRC;

typedef struct _SPWRBK_BASE_TIME_CONFIG {
   ULONG          Count;
   UCHAR          Flags;
   SPWRBK_TM_SRC  TimeSource[SPWRBK_NUM_CHANNELS];
} SPWRBK_BASE_TIME_CONFIG, *PSPWRBK_BASE_TIME_CONFIG;
```

IOCTL_SPWRBK_BASE_GET_TIME_CONFIG

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

IOCTL_SPWRBK_BASE_SET_ENDIAN

Function: Changes the byte-ordering of the DMA data bus.
Input: Big-endian enable (BOOLEAN)
Output: None
Notes: When the input parameter is TRUE, the DMA data-bytes will be configured to use big-endian byte-ordering. When the input parameter is FALSE, the DMA data-bytes will be configured to use little-endian byte-ordering

IOCTL_SPWRBK_BRIDGE_RECONFIG

Function: Finds and configures the Tsi-384 or P17C9X130 PCIe to PCI Bridge if present.
Input: None
Output: None
Notes: Although the bridge should have already been configured when the driver initialized the hardware, this call was added to enhance the DMA data throughput. Occasionally the OS interferes with initial programming at start-up. This call allows the enhanced settings to be applied. New with this release is the addition of the second bridge type.

The IOCTLs defined for the SpWrBkChan driver are described below:

IOCTL_SPWRBK_CHAN_GET_INFO

Function: Returns the driver revision, instance number and transmit and receive FIFO sizes as well as various parameters passed to the channel driver from the base driver.
Input: None
Output: SPWRBK_CHAN_DRIVER_DEVICE_INFO structure
Notes: See the definition of SPWRBK_CHAN_DRIVER_DEVICE_INFO below.

```
// Driver/Device information
typedef struct _SPWRBK_CHAN_DRIVER_DEVICE_INFO {
   UCHAR    DriverRev;
   UCHAR    ChannelNum;
   UCHAR    DesignRev;     // Design revision from base driver
   UCHAR    DesignRevMin;  // Design minor revision from base driver
   UCHAR    DesignType;    // Design type from base driver
   UCHAR    SwitchValue;   // Board user switch value from base driver
   ULONG    InstanceNum;   // Board instance from base driver
   ULONG    TxFifoSize;
   ULONG    RxFifoSize;
} SPWRBK_CHAN_DRIVER_DEVICE_INFO, *PSPWRBK_CHAN_DRIVER_DEVICE_INFO;
```

IOCTL_SPWRBK_CHAN_SET_CONFIG

Function: Specifies the channel control configuration.
Input: SPWRBK_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 SPWRBK_START, SPWRBK_INTS, SPWRBK_DMA_PRMPT, SPWRBK_DMA_STAT and SPWRBK_CHAN_CONFIG below.

```
typedef enum _SPWRBK_START {
   SPWRBK_STOP,           // Channel link not connected
   SPWRBK_ISTRT,          // Channel initiates link
   SPWRBK_ASTRT           // Channel waits for a NULL to be received
} SPWRBK_START, *PSPWRBK_START;

typedef struct _SPWRBK_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
} SPWRBK_INTS, *PSPWRBK_INTS;
```

```
 // Channel DMA priority (use sparingly)
typedef enum _SPWRBK_DMA_PRMPT {
   SPWRBK_NONE,            // No priority
   SPWRBK_READ,            // Read DMA has priority
   SPWRBK_WRITE,           // Write DMA has priority
   SPWRBK_RDWR             // Read and Write DMA have priority
} SPWRBK_DMA_PRMPT, *PSPWRBK_DMA_PRMPT;

typedef enum _SPWRBK_DMA_STAT {
   SPWRBK_BUSY,            // Read and Write DMA both active
   SPWRBK_RD_RDY,          // Read DMA idle
   SPWRBK_WR_RDY,          // Write DMA idle
   SPWRBK_BOTH_RDY         // Read and Write DMA both idle
} SPWRBK_DMA_STAT, *PSPWRBK_DMA_STAT;

typedef struct _SPWRBK_CHAN_CONFIG {
   UCHAR             ClockDivide;   // PLL frequency/(1..16) = I/O bit rate
   SPWRBK_START      StartMode;     // Link start mode (manual or auto)
   SPWRBK_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
   SPWRBK_DMA_PRMPT  DmaPriority;   // DMA preemption control
   SPWRBK_DMA_STAT   DmaStatus;     // DMA status (read-only)
} SPWRBK_CHAN_CONFIG, *PSPWRBK_CHAN_CONFIG;
```

IOCTL_SPWRBK_CHAN_GET_CONFIG

Function: Returns the fields set in the previous call.
Input: None
Output: SPWRBK_CHAN_CONFIG structure
Notes: See the definitions of SPWRBK_START, SPWRBK_INTS, SPWRBK_DMA_PRMPT, SPWRBK_DMA_STAT and SPWRBK_CHAN_CONFIG above.

IOCTL_SPWRBK_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. If the TICK Received interrupt is enabled, the time-code data will be automatically read and the value returned with the ISR status from the interrupt service routine, otherwise, the time-code data must be explicitly read with the read time-code data call.

```
 // 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_SPWRBK_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_SPWRBK_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). If bit 31 is set high, it indicates that either an error condition occurred during the reception of the referenced packet or that the packet was terminated with an EEP. Reading the channel status will indicate whether a connection error was detected.

IOCTL_SPWRBK_CHAN_SET_FIFO_LEVELS

Function: Sets the transmitter almost empty and receiver almost full levels for the channel.
Input: SPWRBK_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 to 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 SPWRBK_CHAN_FIFO_LEVELS below.

```
typedef struct _SPWRBK_CHAN_FIFO_LEVELS {
   ULONG    AlmostFull;
   ULONG    AlmostEmpty;
} SPWRBK_CHAN_FIFO_LEVELS, *PSPWRBK_CHAN_FIFO_LEVELS;
```

IOCTL_SPWRBK_CHAN_GET_FIFO_LEVELS

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

IOCTL_SPWRBK_CHAN_GET_FIFO_COUNTS

Function: Returns the number of data words in the transmit and receive data and packet-length FIFOs.
Input: None
Output: SPWRBK_CHAN_FIFO_COUNTS structure
Notes: There is a 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, for the internal FIFO version, the transmit count can be a maximum of 16,384 32-bit words and the receive count can be a maximum of 16,387 32-bit words. For the -128 version on channel 0 the transmit count can be a maximum of 147,455 32-bit words and the receive count can be a maximum of 147,458 32-bit words. For the -128RX version on channel 0 and 1 the receive count can be a maximum of 147,458 32-bit words. Other FIFOs not mentioned match the internal FIFO version. The TxPktCount and RxPktCount fields can be a maximum of 1023 packet lengths. See the definition of SPWRBK_CHAN_FIFO_COUNTS below.

```
typedef struct _SPWRBK_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
} SPWRBK_CHAN_FIFO_COUNTS, *PSPWRBK_CHAN_FIFO_COUNTS;
```

IOCTL_SPWRBK_CHAN_RESET_FIFOS

Function: Resets one or both FIFOs for the referenced channel.
Input: SPWRBK_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 SPWRBK_FIFO_SEL below.

```
 // Used for FIFO reset call
typedef enum _SPWRBK_FIFO_SEL {
   SPWRBK_TX,
   SPWRBK_RX,
   SPWRBK_BOTH
} SPWRBK_FIFO_SEL, *PSPWRBK_FIFO_SEL;
```

IOCTL_SPWRBK_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_SPWRBK_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_SPWRBK_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_SPWRBK_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_SPWRBK_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_SPWRBK_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_SPWRBK_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. The new field is true if the Status has been updated since it was last read. See the definition of SPWRBK_CHAN_INT_STAT below.

```
typedef struct _SPWRBK_CHAN_INT_STAT {
   ULONG    Status;
   BOOLEAN  New;
} SPWRBK_CHAN_INT_STAT, *PSPWRBK_CHAN_INT_STAT;
```

IOCTL_SPWRBK_CHAN_READ_TIME_CODE

Function: Returns the last time-code received and clears the tick received latched bit.
Input: None
Output: SPWRBK_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. See the definition of SPWRBK_CHAN_TIME_CODE structure below.

```
typedef struct _SPWRBK_CHAN_TIME_CODE {
   UCHAR    Time;
   UCHAR    Flags;
   BOOLEAN  New;
} SPWRBK_CHAN_TIME_CODE, *PSPWRBK_CHAN_TIME_CODE;
```

IOCTL_SPWRBK_CHAN_GET_LINK_STATUS

Function: Reads and returns various quantities related to the performance of the channel's SpaceWire link.
Input: None
Output: SPWRBK_CHAN_LINK_STATUS structure
Notes: Link status values include the number of outstanding FCTs authorized by the receiver, the number of data characters the transmitter is allowed to send as authorized by the remote node's receiver and the current timecode count received by this channel. This register is intended for test and debug only, not for normal operation.

```
typedef struct _SPWR_CHAN_LINK_STATUS {
   UCHAR    FctCount;
   UCHAR    TxCredit;
   UCHAR    TimeData;
} SPWR_CHAN_LINK_STATUS, *PSPWR_CHAN_LINK_STATUS;
```

IOCTL_SPWRBK_CHAN_SET_RPKT_LEN_AFL_LVL

Function: Set a channel's RX packet length FIFO almost full level.
Input: Value of received packet length FIFO almost full level (unsigned short integer)
Output: None
Notes: Due to extended data storage capability, the packet-length FIFO size would sometimes be inadequate to hold the number of packet lengths that were received. To prevent this from happening, the packet-length FIFO almost full signal was added to the flow-control calculation. This register is used to set when this effect occurs.

IOCTL_SPWRBK_CHAN_GET_RPKT_LEN_AFL_LVL

Function: Read and return a channel's RX packet length FIFO almost full level.
Input: None
Output: Value of received packet length FIFO almost full level (unsigned short integer)
Notes: Returns the value set in the previous call.

Write

SpaceWire-BK DMA data is written to the referenced I/O channel device using the write command. Writes are executed using the Windows 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-BK DMA data is read from the referenced I/O channel device using the read command. Reads are executed using the Windows 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.
https://www.dyneng.com/warranty.html

Service Policy

Before returning a product for repair, verify as well as possible that the driver is at fault. The driver has gone through extensive testing, and in most cases it will be “cockpit error” rather than an error with the driver. When you are sure or at least willing to pay to have someone help then call or e-mail and arrange to work with an engineer. We will work with you to determine the cause of the issue.

Support

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

For Service Contact:

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