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

SpWrBase
&
SpWrChan

Driver Documentation

Win32 Driver Model

Manual Revision F
Corresponding Hardware: 10-2004-0803
PMC-SpaceWire Revision C
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 E/F

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

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

Manual Revision F Revised March 3, 2010

This document contains information of proprietary interest to Dynamic Engineering. It has been supplied in confidence and the recipient, by accepting this material, agrees that the subject matter will not be copied or reproduced, in whole or in part, nor its contents revealed in any manner or to any person except to meet the purpose for which it was delivered.

Dynamic Engineering has made every effort to ensure that this manual is accurate and complete. Still, the company reserves the right to make improvements or changes in the product described in this document at any time and without notice. Furthermore, Dynamic Engineering assumes no liability arising out of the application or use of the device described herein.

The electronic equipment described herein generates, uses, and can radiate radio frequency energy. Operation of this equipment in a residential area is likely to cause radio interference, in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

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.

Connection of incompatible hardware is likely to cause serious damage.

Table of Contents

Introduction

The SpWrBase and SpWrChan drivers are Win32 driver model (WDM) device drivers for the PMC-SpaceWire, ccPMC-SpaceWire, PCI-SpaceWire and PC104p-SpaceWire from Dynamic Engineering. These SpaceWire boards have either a Spartan3-1000 Xilinx FPGA or a Spartan3-1500 Xilinx FPGA to implement the PCI interface, 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 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. Channels with external data FIFOs have two 1023 x 32-bit packet-length FIFOs, whereas channels with internal data FIFOs may have either two 1023 x 18-bit or two 1023 x 32-bit packet-length FIFOs depending on the amount of block RAM available in the Xilinx part used.

The SpWrChan driver has been extended to recognize and control an I/O channel using external FIFOs (128 K by 32 bits) as well as the standard internal FIFO (1 K by 32 bits) channel. In order to accommodate this, the data structure fields representing the FIFO counts and programmable almost full/empty levels have been changed to 32-bit values. Also, Tx and Rx FIFO size fields have been added to the channel info structure. When the channel driver initializes, it checks the channel control register. If bits 22 and/or 23 can be cleared, it has detected an external FIFO channel and proceeds to write and read to the programmable almost empty/full registers of the FIFO. Depending on what value is returned, the size of the external FIFO is detected and default values for the Tx almost empty and/or Rx almost full values are written to the FIFO (⅛ and ⅞ of the FIFO size respectively).

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 device.

Notes

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.

Recent Firmware Updates

Revision D: Improved high-speed performance for packet-lengths not divisible by four; extended the maximum packet-lengths to 128 K Bytes and 2 G Bytes depending on Xilinx part and FIFO configuration; added a control bit to reuse a single packet-length and added latched transmit FIFO almost empty and receive FIFO almost full status bits.

Revision E: Corrected low-speed connection problem caused by missing back-to-back FCTs; revised handling of EEPs so that they do not cause a connection error when received and allow sending of an EEP by setting the error bit in the transmit packet-length FIFO (needed by router nodes); revised Xilinx configuration/revision numbering.

Revision F: Added transmit packet-length FIFO count field above transmit data FIFO count field at the same address; replaced transmit packet-length FIFO full status bit with transmitter purge error latched status bit to alert the user of a failure to purge all the data from an aborted transmit packet after a connection error; revised flow-control to be more responsive and efficient.

Driver Installation

There are several files provided in each driver package. These files include SpaceWire.inf, SpWrBase.sys, DDSpWrBase.h, SpWrBaseGUID.h, SpWrChan.sys, DDSpWrChan.h, SpWrChanGUID.h, SpWrTest.exe, and SpWrTest source files.

SpWrBaseGUID.h and SpWrChanGUID.h are C header files that define the device interface identifiers for the drivers. DDSpWrBase.h and DDSpWrChan.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.

SpWrTest.exe is a sample Win32 console applications that makes calls into the SpWrBase/SpWrChan drivers to test each driver call without actually writing any application code. They are not required during driver installation either.

To run SpWrTest, open a command prompt console window and type ***SpWrTest -d0 -?*** to display a list of commands (the SpWrTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***SpWrTest -dn -im*** where ***n*** and ***m*** are the device number and SpWrBase driver ioctl number respectively or ***SpWrTest -cn -im*** where ***n*** and ***m*** are the channel number (0-3) and SpWrChan driver ioctl number respectively.

This test application is intended to test the proper functioning of each driver call, **not** for normal operation.

Windows 2000 Installation

Copy SpaceWire.inf, SpWrBase.sys and SpWrChan.sys to a floppy disk, CD, or some other accessible location.

With the SpaceWire 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 SpaceWire.inf file.
- Select ***Next***.
- Select ***Finish*** to close the ***Found New Hardware Wizard***.

The system should now see the SpaceWire channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

Windows XP Installation

Copy SpaceWire.inf, SpWrBase.sys and SpWrChan.sys to a floppy disk, CD, or some other accessible location.

With the SpaceWire 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 SpaceWire 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 SpWrBaseGUID.h and SpWrChanGUID.h.

Below is example code for opening handles for SpWrBase device *devNum*.

```
// The maximum length of the device name for a given interface
#define MAX_DEVICE_NAME 256
// Handles to the device objects
HANDLE hSpWrBase                          =  INVALID_HANDLE_VALUE;

HANDLE hSpWrChan[SPWR_BASE_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE,
                                            INVALID_HANDLE_VALUE};
// SpaceWire device number
ULONG                             devNum;
// SpaceWire channel handle array index and interface number
ULONG                             chan, i;
// Return status from command
LONG                              status;
// Handle to device interface information structure
HDEVINFO                          hDeviceInfo;
// The actual symbolic link name to use in the createfile
CHAR                              deviceName[MAX_DEVICE_NAME];
// Size of buffer required to get the symbolic link name
DWORD                             requiredSize;
// Interface data structures for this device
SP_DEVICE_INTERFACE_DATA          interfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA pDeviceDetail;
// The base device information structure
SPWR_BASE_DRIVER_DEVICE_INFO     info;
// The channel device information structure
SPWR_CHAN_DRIVER_DEVICE_INFO     cinfo;
// Flag indicating success finding correct device
BOOLEAN                           found = FALSE;

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_SPWR_BASE,
                               NULL,
                               NULL,
                               DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
```

```
if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   printf("**Error: couldn't get class info, (%d)\n", GetLastError());
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

i = 0;
while(!found)
{// Find the interface for device devNum
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_SPWR_BASE,
                                   i,
                                   &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n", i);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
      else
      {
         printf("**Error: couldn't enum device, (%d)\n", status);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }

 // Get the details data to obtain the symbolic link name
   if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                       &interfaceData,
                                       NULL,
                                       0,
                                       &requiredSize,
                                       NULL))
   {
      if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
      {
         printf("**Error: couldn't get interface detail, (%d)\n",
                GetLastError());

         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
   }
```

```
// Allocate a buffer to get detail
  pDeviceDetail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);

  if(pDeviceDetail == NULL)
  {
     printf("**Error: couldn't allocate interface detail\n");
     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     exit(-1);
  }

  pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                      &interfaceData,
                                      pDeviceDetail,
                                      requiredSize,
                                      NULL,
                                      NULL))
  {
     printf("**Error: couldn't get interface detail(2), (%d)\n",
            GetLastError());

     SetupDiDestroyDeviceInfoList(hDeviceInfo);
     free(pDeviceDetail);
     exit(-1);
  }

// Save the name
  lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

// Cleanup search
  free(pDeviceDetail);

// Open driver - Create the handle to the device
  hSpWrBase = CreateFile(deviceName,
                         GENERIC_READ     | GENERIC_WRITE,
                         FILE_SHARE_READ | FILE_SHARE_WRITE,
                         NULL,
                         OPEN_EXISTING,
                         NULL,
                         NULL);

  if(hSpWrBase == INVALID_HANDLE_VALUE)
  {
     printf("**Error: couldn't open %s, (%d)\n", deviceName,
            GetLastError());

     exit(-1);
  }
```

```
// Read info
   if(!DeviceIoControl(hSpWrBase,
                       IOCTL_SPWR_BASE_GET_INFO,
                       NULL,
                       0,
                       &info,
                       sizeof(info),
                       &length,
                       NULL))
   {
      printf("IOCTL_SPWR_BASE_GET_INFO failed:  %d\n", GetLastError());
      exit(-1);
   }

   if(info.InstanceNumber == devNum)
      found = TRUE;
   else
      i++;
}

SetupDiDestroyDeviceInfoList(hDeviceInfo);

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_SPWR_CHAN,
                               NULL,
                               NULL,
                               DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);

if(hDeviceInfo == INVALID_HANDLE_VALUE)
{
   status = GetLastError();
   printf("**Error: couldn't get class info, (%d)\n", status);
   exit(-1);
}

interfaceData.cbSize = sizeof(interfaceData);

i    = 0;
chan = 0;

while(chan < SPWR_BASE_NUM_CHANNELS)
{// Find the interface for device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                          (LPGUID)&GUID_DEVINTERFACE_SPWR_CHAN,
                                   i,
                                  &interfaceData))
   {
      status = GetLastError();
      if(status == ERROR_NO_MORE_ITEMS)
      {
         printf("**Error: couldn't find device(no more items), (%d)\n", i);
         SetupDiDestroyDeviceInfoList(hDeviceInfo);
         exit(-1);
      }
```

```
    else
    {
       printf("**Error: couldn't enum device, (%d)\n", status);
       SetupDiDestroyDeviceInfoList(hDeviceInfo);
       exit(-1);
    }
  }

// Get the details data to obtain the symbolic link name
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                      &interfaceData,
                                       NULL,
                                       0,
                                      &requiredSize,
                                       NULL))
  {
    if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
    {
       printf("**Error: couldn't get interface detail, (%d)\n",
              GetLastError());

       SetupDiDestroyDeviceInfoList(hDeviceInfo);
       exit(-1);
    }
  }

// Allocate a buffer to get detail
  pDeviceDetail =
    (PSP_DEVICE_INTERFACE_DETAIL_DATA)malloc(requiredSize);
  if(pDeviceDetail == NULL)
  {
    printf("**Error: couldn't allocate interface detail\n");
    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    exit(-1);
  }

  pDeviceDetail->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

// Get the detail info
  if(!SetupDiGetDeviceInterfaceDetail(hDeviceInfo,
                                       &interfaceData,
                                       pDeviceDetail,
                                       requiredSize,
                                       NULL,
                                       NULL))
  {
    printf("**Error: couldn't get interface detail(2), (%d)\n",
           GetLastError());

    SetupDiDestroyDeviceInfoList(hDeviceInfo);
    free(pDeviceDetail);
    exit(-1);
  }
```

```
  // Save the name
    lstrcpyn(deviceName, pDeviceDetail->DevicePath, MAX_DEVICE_NAME);

  // Cleanup search
    free(pDeviceDetail);

  // Open driver - Create the handle to the device
    hSpWrChan[chan] = CreateFile(deviceName,
                                 GENERIC_READ     | GENERIC_WRITE,
                                 FILE_SHARE_READ | FILE_SHARE_WRITE,
                                 NULL,
                                 OPEN_EXISTING,
                                 NULL,
                                 NULL);

    if(hSpWrChan[chan] == INVALID_HANDLE_VALUE)
    {
       printf("**Error: couldn't open %s, (%d)\n",
              deviceName, GetLastError());
       SetupDiDestroyDeviceInfoList(hDeviceInfo);
       exit(-1);
    }

    if(!DeviceIoControl(hSpWrChan[chan],
                        IOCTL_SPWR_CHAN_GET_INFO,
                        NULL,
                        0,
                        &cinfo,
                        sizeof(cinfo),
                        &length,
                        NULL) )
    {
       printf("IOCTL_SPWR_CHAN_GET_INFO failed:  %d\n", GetLastError());
       exit(-1);
    }

    if(cinfo.InstanceNumber / 4 == devNum &&
       cinfo.InstanceNumber % 4 == chan)
    {
       chan++;
    }

    i++;
 }
```

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 Driver version, Xilinx revision, Switch value, Instance number, and PLL ID.
Input: None
Output: SPWR_BASE_DRIVER_DEVICE_INFO structure
Notes: The current base driver version is 1.4. The switch value is the configuration of the on-board dip-switch that is 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 DDSpWrBase.h for the definition of SPWR_BASE_DRIVER_DEVICE_INFO.

IOCTL_SPWR_BASE_LOAD_PLL_DATA

Function: Loads the internal registers of the PLL.
Input: SPWR_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_SPWR_BASE_READ_PLL_DATA

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

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).

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, transmit and receive FIFO sizes, maximum packet-length of the referenced channel and a flag indicating whether the channel has enhanced control/status registers.
Input: None
Output: SPWR_CHAN_DRIVER_DEVICE_INFO structure
Notes: The current driver version is 1.6. The enhancements to the control/status registers are: A control bit to enable repeated transmit packet-length reuse and latched almost empty/full FIFO status bits. These were added to the control/status registers in SpWrChan ver.1.4. See the definition of SPWR_CHAN_DRIVER_DEVICE_INFO in the DDSpWrChan.h header file.

IOCTL_SPWR_CHAN_SET_CONFIG

Function: Writes a configuration value to the channel control register.
Input: Value of channel control register (unsigned long integer)
Output: None
Notes: See DDSpWrChan.h for the relevant channel control bit definitions. Only the bits in CHAN_CNTRL_MASK can be controlled by this call.

IOCTL_SPWR_CHAN_GET_CONFIG

Function: Returns the channel's control configuration.
Input: None
Output: Value of the channel control register (unsigned long integer)
Notes: Returns the values of the bits in CHAN_CNTRL_READ_MASK.

IOCTL_SPWR_CHAN_GET_STATUS

Function: Returns the channel's status value and clears the latched bits.
Input: None
Output: Value of channel status register (unsigned long integer)
Notes: The latched bits in CHAN_STAT_LATCH_MASK will be cleared if they are set when the status is read. Even though CHAN_STAT_TICK_RCVD is latched, it is not cleared by this call. It will be cleared in the ISR if the CHAN_CNTRL_TICK_INTEN bit is set or by the IOCTL_SPWR_CHAN_READ_TIME_CODE call. In firmware version F a transmit packet-length FIFO count was added and the CHAN_STAT_TX_PKTFL status bit was replaced by the CHAN_STAT_TX_PURGERR latched status bit.

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 17 high (bit 32 for 2 G Byte packet-length channels) causes the transmitted packet to be terminated with an EEP.

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: Only bits 16-0 (30-0 for 2 G Byte packet-length channels) are used for the packet-length (maximum of 128 K Bytes (2 G Bytes)). Bit 17 (bit 32 for 2 G Byte packet-length channels) 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.

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, receive and transmit 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, for the internal FIFO channels, 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. For external FIFO channels, the transmit count can be a maximum of 131,073 32-bit words and the receive count can be a maximum of 131,076 32-bit words. In channel driver ver.1.6 (firmware version F) a transmit packet-length FIFO count was added and this field was added to the SPWR_CHAN_FIFO_COUNTS structure. If the referenced hardware does not support this function, the TxPktCount field will be set to 0xFFFF otherwise it will reflect the number of values in the transmit packet-length FIFO (0 to 0x3FF).

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.

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 the 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 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.

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.

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

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