**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

# PmcWiz
# &
# WizChan

## Driver Documentation

### Win32 Driver Model

Revision B
Corresponding Hardware: Revision A/B
10-2005-0501/0502
Corresponding Firmware: Revision B

**PmcWiz, WizChan**
WDM Device Drivers for the
PMC-Wizard 2-Channel
WizardLink Interface

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

Manual Revision B. Revised February 24, 2012

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 PmcWiz and WizChan drivers are Win32 driver model (WDM) device drivers for the PMC-Wizard from Dynamic Engineering. The PMC-Wizard board has a Spartan3-1500 Xilinx FPGA to implement the PCI interface, FIFOs and protocol control and status for two WizardLink channels. Each channel has two 4k x 32-bit data FIFOs for data transmission and reception.

When the PMC-Wizard is recognized by the PCI bus configuration utility it will start the PmcWiz driver. The PmcWiz driver enumerates the two channels and creates a separate WizChan device object for each. This allows the I/O channels to be totally independent while the base driver controls the device items that are common to both channels. 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 PMC-Wizard user manual (also referred to as the hardware manual).

## Driver Installation

There are several files provided in each driver package. These files include PmcWiz.sys, PmcWiz.inf, DDPmcWiz.h, PmcWizGUID.h, WizChan.sys, WizChan.inf, DDWizChan.h, WizChanGUID.h, PmcWizTest.exe, and PmcWizTest source files.

## Windows 2000 Installation

Copy PmcWiz.inf, WizChan.inf, PmcWiz.sys and WizChan.sys to a floppy disk, CD, or some other accessible location.

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

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

## Windows XP Installation

Copy PmcWiz.inf, WizChan.inf, PmcWiz.sys and WizChan.sys to a floppy disk, CD, or some other accessible location.

With the PmcWizard 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 PmcWizard channels and reopen the ***New Hardware Wizard.*** Proceed as above for each channel as necessary.

DDPmcWiz.h and DDWizChan.h are C header files that define the Application Program Interface (API) to the drivers. PmcWizGUID.h and WizChanGUID.h are C header files that define the device interface identifiers for the PmcWiz and WizChan 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.

The PmcWizTest.exe file is a sample Win32 console application that makes calls into the PmcWiz/WizChan drivers to test each driver call without actually writing any application code. It is not required during the driver installation.

To run PmcWizTest.exe, open a command prompt console window and type ***PmcWizTest -d0 -?*** to display a list of commands (the PmcWizTest.exe file must be in the directory that the window is referencing). The commands are all of the form ***PmcWizTest -dn -im*** where ***n*** and ***m*** are the device number and driver PmcWiz ioctl number respectively or ***PmcWizTest -cn -im*** where ***n*** and ***m*** are the channel number and WizChan driver ioctl number respectively. This application is intended to test the proper functioning of the driver calls, not for normal operation.

## 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 a globally unique identifier (GUID), which is defined in PmcWizGUID.h and WizChanGUID.h.

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

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

HANDLE hWizChan[PMC_WIZ_NUM_CHANNELS] = {INVALID_HANDLE_VALUE,
                                         INVALID_HANDLE_VALUE};
// PmcWizard device number
ULONG                            devNum;
// PmcWizard 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 reguired 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
PMC_WIZ_DRIVER_DEVICE_INFO       info;
// The channel device information structure
WIZ_CHAN_DRIVER_DEVICE_INFO      cinfo;
// Flag indicating success finding correct device
BOOLEAN                          found = FALSE;

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_PMC_WIZ,
                               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_PMC_WIZ,
                                   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
   hPmcWiz = CreateFile(deviceName,
                        GENERIC_READ    | GENERIC_WRITE,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        NULL,
                         NULL);

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

      exit(-1);
   }

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

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

SetupDiDestroyDeviceInfoList(hDeviceInfo);

hDeviceInfo = SetupDiGetClassDevs(
                       (LPGUID)&GUID_DEVINTERFACE_WIZ_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 < PMC_WIZ_NUM_CHANNELS)
{// Find the interface for device
   if(!SetupDiEnumDeviceInterfaces(hDeviceInfo,
                                   NULL,
                           (LPGUID)&GUID_DEVINTERFACE_WIZ_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
  hWizChan[chan] = CreateFile(deviceName,
                              GENERIC_READ    | GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE_WRITE,
                              NULL,
                              OPEN_EXISTING,
                              NULL,
                              NULL);

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

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

   if(cinfo.InstanceNumber / 2 == devNum &&
      cinfo.InstanceNumber % 2 == 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 PmcWiz driver are described below:**

### IOCTL_PMC_WIZ_GET_INFO

***Function:*** Returns the Driver version, Xilinx flash revision, Switch value, Instance number and PLL device ID.
***Input:*** None
***Output:*** PMC_WIZ_DRIVER_DEVICE_INFO structure
***Notes:*** Switch value is the configuration of the 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. The PLL ID is the device address of the PLL. This value, which is set at the factory, is usually 0x69 but may also be 0x6A. See DDPmcWiz.h for the definition of PMC_WIZ_DRIVER_DEVICE_INFO.

### IOCTL_PMC_WIZ_LOAD_PLL_DATA

***Function:*** Loads the internal registers of the PLL.
***Input:*** PMC_WIZ_PLL_DATA structure
***Output:*** None
***Notes:*** The PMC_WIZ_PLL_DATA structure has only one field: Data – an array of 40 bytes containing the data to write.

### IOCTL_PMC_WIZ_READ_PLL_DATA

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

### IOCTL_PMC_WIZ_GET_INT_STATUS

***Function:*** Returns the interrupt status.
***Input:*** None
***Output:*** Interrupt status value (unsigned long integer)
***Notes:*** This command returns the individual interrupt status bits and the interrupt active status bit.  See the bit definitions in DDPmcWiz.h for information on interpreting this value.

### IOCTL_PMC_WIZ_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.

### IOCTL_PMC_WIZ_ENABLE_INTERRUPT

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

### IOCTL_PMC_WIZ_DISABLE_INTERRUPT

***Function:*** Disables the master interrupt.
***Input:*** None
***Output:*** None
***Notes:*** This call is used when local interrupt processing is no longer desired.

### IOCTL_PMC_WIZ_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 master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

### IOCTL_PMC_WIZ_GET_ISR_STATUS

***Function:*** Returns the interrupt status that was 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 serviced.

**The IOCTLs defined for the WizChan driver are described below:**

### IOCTL_WIZ_CHAN_GET_INFO

***Function:*** Returns the Driver version and Instance number.
***Input:*** None
***Output:*** WIZ_CHAN_DRIVER_DEVICE_INFO structure
***Notes:*** See DDWizChan.h for the definition of WIZ_CHAN_DRIVER_DEVICE_INFO.

### IOCTL_WIZ_CHAN_RESET_FIFOS

***Function:*** Resets the channel's FIFOs.
***Input:*** None
***Output:*** None
***Notes:*** Resets the Tx and Rx FIFOs for the referenced channel.

### IOCTL_WIZ_CHAN_SET_CONFIG

***Function:*** Writes to the channel's control register.
***Input:*** Value of the control register (unsigned long integer)
***Output:*** None
***Notes:*** Only the bits in the CNTRL_MASK are controlled by this command. See the bit definitions in DDWizChan.h for information on determining this value.

### IOCTL_WIZ_CHAN_GET_CONFIG

***Function:*** Returns the configuration of the control register.
***Input:*** None
***Output:*** Value of the control register (unsigned long integer)
***Notes:*** The return value includes the bits in CNTRL_MASK and CNTRL_DMA_WREN, CNTRL_DMA_RDEN and CNTRL_MINTEN. This command is used mainly for testing.

### IOCTL_WIZ_CHAN_GET_STATUS

***Function:*** Returns the channel's status word and clears the latched bits.
***Input:*** None
***Output:*** Value of the channel's status register (unsigned long integer)
***Notes:*** See DDWizChan.h for the status bit definitions.

### IOCTL_WIZ_CHAN_SET_SYNC_PATTERN

***Function:*** Sets the channel's receive sync pattern.
***Input:*** Sync pattern to detect (unsigned long integer)
***Output:*** None
***Notes:*** In order to detect the start of a message, the receiver looks for this pattern to match the four-byte header.

### IOCTL_WIZ_CHAN_GET_SYNC_PATTERN

***Function:*** Returns the channel's receive sync pattern.
***Input:*** None
***Output:*** (unsigned long integer)
***Notes:*** Returns the pattern written by the previous call.

### IOCTL_WIZ_CHAN_SET_FIFO_LEVELS

***Function:*** Sets the channel's receiver almost full and transmitter almost empty levels.
***Input:*** WIZ_CHAN_FIFO_LEVELS structure
***Output:*** None
***Notes:*** The FIFO levels are used to determine at what data count the TX almost empty and RX almost full status bits are asserted. See DDWizChan.h for the definition of WIZ_CHAN_FIFO_LEVELS.

### IOCTL_WIZ_CHAN_GET_FIFO_LEVELS

***Function:*** Returns the channel's receiver almost full and transmitter almost empty levels.
***Input:*** None
***Output:*** WIZ_CHAN_FIFO_LEVELS structure
***Notes:*** Returns the values written in the previous call. See DDWizChan.h for the definition of WIZ_CHAN_FIFO_LEVELS.

### IOCTL_WIZ_CHAN_WRITE_FIFO

***Function:*** Writes a data word to the channel's transmit FIFO.
***Input:*** FIFO data word (unsigned long integer)
***Output:*** None
***Notes:***

### IOCTL_WIZ_CHAN_READ_FIFO

***Function:*** Reads a data word from the channel's receive FIFO.
***Input:*** None
***Output:*** FIFO data word (unsigned long integer)
***Notes:***

### IOCTL_WIZ_CHAN_GET_FIFO_COUNTS

***Function:*** Returns the number of data words in the transmit and receive FIFOs.
***Input:*** None
***Output:*** WIZ_CHAN_FIFO_COUNTS structure
***Notes:*** Returns the current FIFO data counts. See DDWizChan.h for the definition of WIZ_CHAN_FIFO_COUNTS.

### IOCTL_WIZ_CHAN_SET_IDLE_CONFIG

***Function:*** Sets the channel's transmit idle configuration.
***Input:*** WIZ_CHAN_IDLE_CONFIG structure
***Output:*** None
***Notes:*** Between messages the transmitter will insert a specified number of two-byte idle words. This call specifies both the value and number of these idle words. See DDWizChan.h for the definition of WIZ_CHAN_IDLE_CONFIG.

### IOCTL_WIZ_CHAN_GET_IDLE_CONFIG

***Function:*** Returns the channel's transmit idle configuration.
***Input:*** None
***Output:*** WIZ_CHAN_IDLE_CONFIG structure
***Notes:*** Returns the values written in the previous call. See DDWizChan.h for the definition of WIZ_CHAN_IDLE_CONFIG.

### IOCTL_WIZ_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_WIZ_CHAN_ENABLE_INTERRUPT

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

### IOCTL_WIZ_CHAN_DISABLE_INTERRUPT

***Function:*** Disables the master interrupt.
***Input:*** None
***Output:*** None
***Notes:*** This call is used when local interrupt processing is no longer desired.

### IOCTL_WIZ_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 master interrupt is enabled. This IOCTL is used for development, to test interrupt processing.

### IOCTL_WIZ_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 for the last interrupt serviced.

### Write

PMC-Wizard DMA data is written to the device using the write command.  Writes are executed using the Win32 function WriteFile() and passing in the handle to the device opened with CreateFile(), a pointer to a pre-allocated buffer containing the data to be written, an unsigned long integer that represents the size of that buffer in bytes, a pointer to an unsigned long integer to contain the number of bytes actually written, and a pointer to an optional Overlapped structure for performing asynchronous IO.

### Read

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

## 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 St. Suite C
Santa Cruz, CA 95060
831-457-8891
831-457-4793 Fax
support@dyneng.com