Expert I/O
Software User Manual

Revision 2 | January 6, 2010

Dajac Inc.

www.dajac.com

sales@dajac.com

Contents

Software

The software distribution contains an application programming interface (API), source code examples and demo applications. The API is within a dynamic link library (DLL) and allows the user to control every aspect of the Expert I/O. Since a DLL is used, the API is accessible from any programming or script language that supports DLL's. A few examples are C, C++, C#, Visual Basic, Delphi, Java, Python, PHP, etc.

All functions use the __stdcall calling convention.

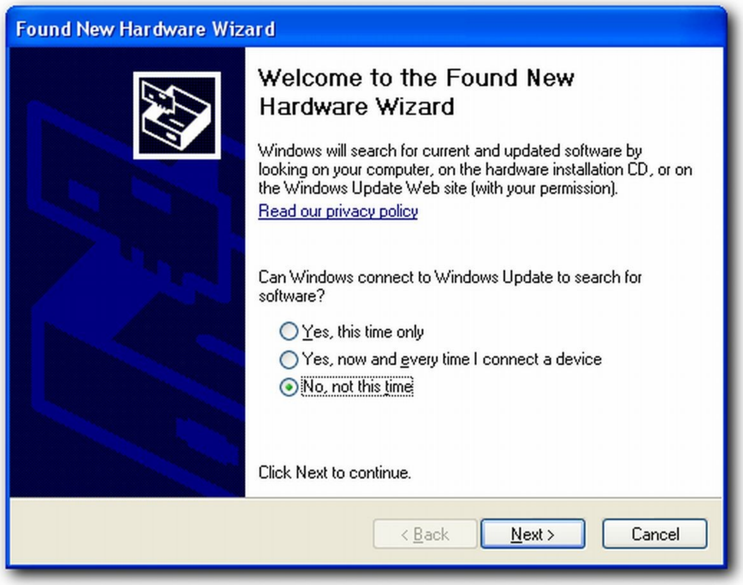
You may freely distribute the API library (eio.dll) with any application you create. See the license agreement for complete licensing information.

A National Instruments LabView driver is also included in the SDK.

Installation

To install, run setup.exe. This will install the driver, the API, examples and documentation.

Power the Expert I/O and attach the USB cable from the Expert I/O to the PC. The following MS Windows message will appear.



Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

Windows will search for current and updated software by looking on your computer, on the hardware installation CD, or on the Windows Update Web site (with your permission).

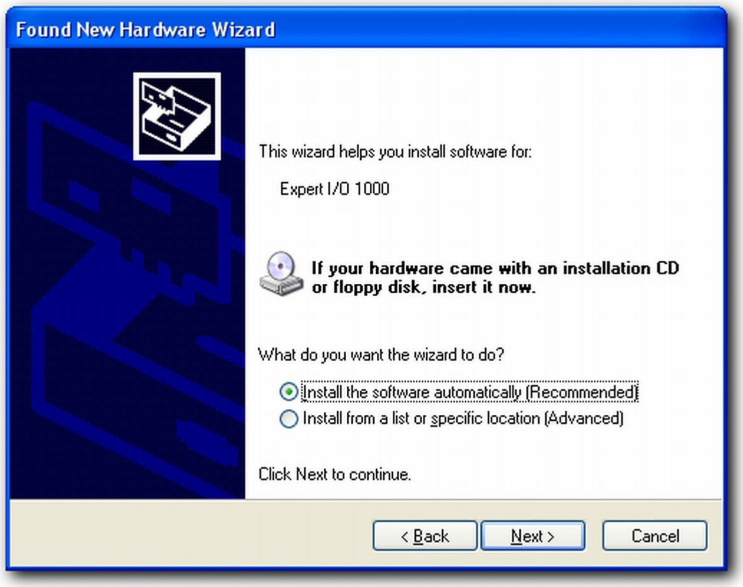
Read our privacy policy

Can Windows connect to Windows Update to search for software?

- ○ Yes, this time only
- ○ Yes, now and every time I connect a device
- [✓ No, not this time]

Click Next to continue.

Select “No, not this time” and click Next.


Found New Hardware Wizard

This wizard helps you install software for:

Expert I/O 1000

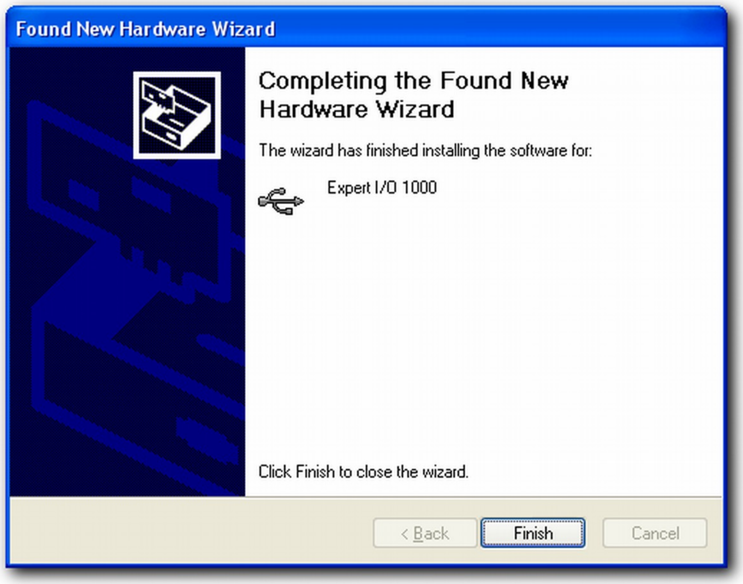
If your hardware came with an installation CD or floppy disk, insert it now.

What do you want the wizard to do?

[✓ Install the software automatically (Recommended)]
Install from a list or specific location (Advanced)

Click Next to continue.

Select “Install the software automatically (Recommended)” and click Next.


Found New Hardware Wizard

Completing the Found New Hardware Wizard

The wizard has finished installing the software for:

Expert I/O 1000

Click Finish to close the wizard.

The Expert I/O is now ready to use.

Each time a new Expert I/O is attached, MS Windows will display the Found New Hardware Wizard. Follow the steps above for each occurrence. Once the Found New Hardware Wizard is completed, it will not be shown again for that Expert I/O.

National Instruments LabView Driver

To use the LabView driver, you must first install the SDK as described above. During the SDK installation, the LabView driver will be placed in the directory you chose for the SDK install. Simply unzip the LabView driver into your LabView instr.lib directory and restart LabView. From within LabView, you can access full documentation and examples.

Application Programming Interface

The following sections describe the API in detail.

Initialization

These functions are used for initialization and cleanup. Declarations are in eio_init.h.

eioOpen

Syntax:

```
int eioOpen( void )
```

Description:

Initialize the API. Call this function before communicating with any Expert I/O units.

Parameters:

none

Return Value:

`eioSUCCESS`: No problems.

`eioERROR`: An error occured.

eioClose

Syntax:

```
int eioClose( void )
```

Description:

Call this function when it is no longer necessary to communicate with any Expert I/O units.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occured.

eioGetDescriptors

Syntax:

```
int __stdcall eioGetDescriptors( eioTDescriptorGroup *DescriptorGroup )
```

Description:

Get descriptors for all the Expert I/O units attached to all the USB buses. This function searches all the USB buses and returns a descriptor for each Expert I/O device found. Descriptors provide information about the devices found.

Parameters:

DescriptorGroup

This is a pointer to a structure that holds a descriptor for every Expert I/O device found. Memory for each descriptor is allocated by this library. The memory will be released when eioClose() is called. The descriptor memory is owned by this library and should not be modified by outside code.

The `DescriptorGroup` structure must be allocated prior to calling this function.

`eioTDescriptorGroup` is defined in eio_common.h as follows.

```
typedef struct
{
  int NumDescriptors;
  eioTDescriptor **Descriptors;
} eioTDescriptorGroup;
```

`NumDescriptors` is the number of descriptors contained in the group.

`Descriptors` is an array of the descriptors and `eioTDescriptor` is a structure defined in eio_common.h.

```
typedef struct
{
  unsigned int SerialNumber;
  unsigned short ModelId;
  char ModelString[MAX_MODEL_STR_LEN];
} eioTDescriptor;
```

`SerialNumber` is the serial number of the Expert I/O.

`ModelId` is the model ID of the Expert I/O.

`ModelString` is a NULL terminated string version of the model ID. This is useful when the model ID must be shown to the user.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occured.

eioClaimInterface

Syntax:

```
int eioClaimInterface( eioTDescriptor *Descriptor, eioTHandle *Handle )
```

Description:

Claim the interface of the device with the given descriptor. The device's interface must be claimed before the device can be accessed. Once a claimed interface is no longer needed, it must be released. If a device's interface has been claimed, it can't be claimed again until it has been released.

Parameters:

Descriptor

The descriptor for the device whose interface is to be claimed.

Handle

Upon successful return, this will be a handle to the specified device. If unsuccessful, it will be NULL.

Return:

`eioSUCCESS`: No problems.

`eioDEVICE_NOT_FOUND`: Could not find the requested device.

`eioUNABLE_TO_CLAIM_INTERFACE`: The device was found, but its interface could not be claimed. A couple possible causes are not enough memory or the interface has already been claimed.

eioClaimInterfaceEz

Syntax:

```
int eioClaimInterfaceEz( unsigned short ModelId, unsigned int
SerialNumber, eioTHandle *Handle )
```

Description:

This function is provided as a simpler alternative to `eioClaimInterface`. It allows you to claim an interface by only providing model ID and serial number. Using this method, you don't need to scan the USB for descriptors.

The device's interface must be claimed before the device can be accessed. Once a claimed interface is no longer needed, it must be released. If a device's interface has been claimed, it can't be claimed again until it has been released.

Parameters:

<u>*ModelId*</u>

The model ID for the device whose interface is to be claimed. See the `eioMID_EIOxx` defines in `eio_prod_info.h` for a list of valid model ID's. The model ID is also included in the hardware manual.

<u>*SerialNumber*</u>

The serial number of the device whose interface is to be claimed.

<u>*Handle*</u>

Upon successful return, this will be a handle to the specified device. If unsuccessful, it will be NULL.

Return:

`eioSUCCESS`: No problems.

`eioDEVICE_NOT_FOUND`: Could not find the requested device.

`eioUNABLE_TO_CLAIM_INTERFACE`: The device was found, but its interface could not be claimed. A couple possible causes are not enough memory or the interface has already been claimed.

eioReleaseInterface

Syntax:

```
int eioReleaseInterface( eioTHandle Handle )
```

Description:

Release a device's interface. Call this function after a device's interface has been claimed and is not longer needed. Once freed, the interface is available to be claimed again.

Parameters:

<u>*Handle*</u>

A handle to the device that owns the interface to be released.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occured.

Analog Input (ADC)

Use the analog input interface to read analog voltages. Declarations for this interface are in eio_adc.h.

eioAdcReadCount

Syntax:

```
int eioAdcReadCount( eioTHandle Handle, int Input, unsigned int *Count )
```

Description:

Read the ADC count for the given input.

Parameters:

Handle

The handle of the Expert I/O device to query.

Input

The input to read.

Count

Upon return, this will hold the ADC count of the given input.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: A problem occurred.

Analog Output (DAC)

The analog output interface allows controlling the analog output voltages. Declarations for this interface are in eio_dac.h.

The DAC digital inputs are double buffered. When controlling a DAC, control values can be loaded into the input register and then shifted to the output register to make the control value active at the output. Alternatively, control values can be loaded into the input and output registers at the same time.

Output voltages are set by loading the DAC registers with values relative to the number of bits supported by the DAC and the current output range setting. For example, loading 128 into an 8-bit DAC whose range is 0V to 5V, results in a 2.5V level at the output. If the range were -5V to +5V, then the output voltage would be 0V. As can be seen, zero count always results in the minimum output voltage and the maximum count always results in the maximum output voltage.

eioDacInOut

Syntax:

```
int eioDacInOut( eioTHandle Handle, int Dac, unsigned int Count )
```

Description:

Load a count into both, the input and the output registers simultaneously. The new voltage will appear at the analog output immediately.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port to act upon. Use `eioDAC_ALL` to change all analog outputs to the given value.

Count

The value that will be loaded into the input and output registers.

Return:

`eioSUCCESS` => No problems.

`eioERROR` => An error occurred.

eioDacIn

Syntax:

```
int eioDacIn( eioTHandle Handle, int Dac, unsigned int Count )
```

Description:

Load a count into the input register. The new voltage will not appear at the analog output until the count is transferred to the output register.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port to act upon. Use `eioDAC_ALL` to act upon all analog outputs.

Count

The value that will be loaded into the input register.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacInToOut

Syntax:

```
int eioDacInToOut( eioTHandle Handle, int Dac )
```

Description:

Transfer the value in the input register to the output register. The transferred value becomes effective at the analog output immediately.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port to act upon. Use `eioDAC_ALL` to act upon all analog outputs.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacGetCount

Syntax:

```
int eioDacGetCount( eioTHandle Handle, int Dac, unsigned int *Count )
```

Description:

Read the value currently in the output register.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port of interest.

Count

Upon return, this will contain the value currently in the output register.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacSetClear

Syntax:

```
int eioDacSetClear( eioTHandle Handle, int Dac, eioTBool Clear )
```

Description:

Each analog output can be cleared. When cleared, the output is immediately forced to ground. The output will remain at ground until the clear status is deactivated.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port to act upon. Use `eioDAC_ALL` to act upon all analog outputs.

Clear

Set to `eioTRUE` to activate the clear condition. The analog output and all internal registers immediately go to zero. The analog output will remain zero until the clear condition is removed.

Set to `eioFALSE` to release the clear condition. After being released, the analog output can once again be set to values other than zero.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacGetClear

Syntax:

```
int eioDacGetClear( eioTHandle Handle, int Dac, eioTBool *Clear )
```

Description:

Get the current value of the analog output's clear feature.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port of interest.

Clear

Upon return, this will contain the current setting of the clear feature. The clear feature can be either `eioTRUE` (active) or `eioFALSE` (inactive).

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacSetRange

Syntax:

```
int eioDacSetRange( eioTHandle Handle, int Dac, int Range,
  unsigned int Count )
```

Description:

Each analog output can be set to one of several output ranges. This function sets a new range for the specified analog output.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port of interest. Use `eioDAC_ALL` to act upon all analog outputs.

Range

The range can be set as follows.

Setting	Minimum (V)	Maximum (V)
`eioDAC_P5`	0	5
`eioDAC_P10`	0	10
`eioDAC_N5_P5`	-5	5
`eioDAC_N10_P10`	-10	10
`eioDAC_2N5_2P5`	-2.5	2.5
`eioDAC_2N5_7P5`	-2.5	7.5

Count

When the range is set, this count will be loaded into the input and output registers.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDacGetRange

Syntax:

```
int eioDacGetRange( eioTHandle Handle, int Dac, int *Range )
```

Description:

Get the current range of the specified analog output.

Parameters:

Handle

The handle of the Expert I/O device.

Dac

The analog output port of interest.

Range

Upon return, this will contain the current range setting. The description of the `eioDacSetRange` function lists the allowable range settings.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

Digital I/O

The digital I/O interface declarations are in eio_dio.h. Using this interface, it is possible to read the digital inputs and control the digital outputs. Additionally, this interface is used to control the programmable pull feature that allows programming pull ups and pull downs on the I/O lines.

eioDioGet

Syntax:

```
int eioDioGet( eioTHandle Handle, int Port, unsigned char *PinLevels )
```

Description:

Read a digital I/O port. This works for inputs as well as outputs.

Parameters:

Handle

The handle of the Expert I/O device.

Port

The port of interest.

PinLevels

Upon return, this will contain the pin levels of the specified port. Each bit corresponds to a digital I/O pin. A bit value of one means the pin is at a high voltage and a bit value of zero means it is at ground.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDioSet

Syntax:

```
int eioDioSet( eioTHandle Handle, int Port, unsigned char PinLevels )
```

Description:

Set the output voltage levels of the specified port.

Parameters:

Handle

The handle of the Expert I/O device.

Port

The port of interest.

PinLevels

Each pin's voltage level will be set according to the value of this parameter. Each bit in `PinLevels` corresponds to a digital output pin. A bit value of one means the pin will be set to a high voltage and a bit value of zero means it will be set to ground.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDioGetPull

Syntax:

```
int eioDioGetPull( eioTHandle Handle, int Port, int *PullLevel )
```

Description:

Get the pull setting of the given port.

Parameters:

Handle

The handle of the Expert I/O device.

Port

The port of interest.

PullLevel

Upon successful return, this will contain the level to which the port is being pulled. See `eioDioSetPull` for valid pull levels.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioDioSetPull

Syntax:

```
int eioDioSetPull( eioTHandle Handle, int Port, int PullLevel )
```

Description:

Set the pull level of the given port.

Parameters:

Handle

The handle of the Expert I/O device.

Port

The port of interest.

PullLevel

The port's pull feature will be set to this value. Valid pull levels are shown below.

Pull Level	Description
`eioDIO_PULL_B`	Pull is to +B.
`eioDIO_PULL_B1`	Pull is to +B1.
`eioDIO_PULL_3_3`	Pull is to 3.3V.
`eioDIO_PULL_GND`	Pull is to ground. This is not a valid setting for the digital outputs.
`eioDIO_PULL_NONE`	Pull is disabled. This is not a valid setting for the digital outputs.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

EEPROM

Use the EEPROM interface to access the EEPROM. This interface is declared in eio_eeprom.h.

eioEepromRead

Syntax:

```
unsigned int eioEepromRead( eioTHandle Handle, unsigned int Address,
  unsigned int NumBytes, void *Destination )
```

Description:

Read data from the EEPROM.

Parameters:

Handle

The handle of the Expert I/O device.

Address

The EEPROM address at which to read.

NumBytes

The number of data bytes to read.

Destination

The data read from the EEPROM will be stored here. The calling code must ensure that enough memory has been allocated to hold `NumBytes` of data.

Return:

The number of bytes read. This will be less than `NumBytes` if an error occurred or if there was an attempt to read past the last address in the EEPROM.

eioEepromWrite

Syntax:

```
unsigned int eioEepromWrite( eioTHandle Handle, unsigned int Address,
  unsigned int NumBytes, void *Source )
```

Description:

Write data to the eeprom.

Parameters:

Handle

The handle of the Expert I/O device.

Address

The EEPROM address at which to write.

NumBytes

The number of data bytes to write.

Source

The data to write into the EEPROM.

Return:

The number of bytes written. This will be less than `NumBytes` if an error occurred or if there was an attempt to write past the last address in the EEPROM.

Motor Control

This interface is used to access the motor controllers. Declarations are in eio_mc.h.

eioMcSetVelocity

Syntax:

```
int eioMcSetVelocity( eioTHandle Handle, int Controller, int Velocity )
```

Description:

Set the motor velocity.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest. Use `eioMC_ALL` to simultaneously set the velocity of all motors to the same value.

Velocity

This is the PWM pulse width count. The maximum count is `MC_MAX_PULSEWIDTH`. Positive velocity rotates the motor clockwise. Negative velocity rotates the motor counterclockwise.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioMcGetVelocity

Syntax:

```
int eioMcGetVelocity( eioTHandle Handle, int Controller, int *Velocity )
```

Description:

Get the current motor velocity.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest.

Velocity

Upon return, this contains the current PWM pulse width count. See `eioMcSetVelocity` for interpretation of this value.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioMcSetBrakeState

Syntax:

```
int eioMcSetBrakeState( eioTHandle Handle, int Controller,
  int BrakeState )
```

Description:

Set the motor controller brake state. The motor stops immediately when the brake is enabled and continues at the current velocity setting when the brake is disabled.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest. Use `eioMC_ALL` to simultaneously set the brake of all motors to the same state.

BrakeState

The desired state of the motor controller brake. Valid values are `eioON` and `eioOFF`.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioMcGetBrakeState

Syntax:

```
int eioMcGetBrakeState( eioTHandle Handle, int Controller,
  int *BrakeState )
```

Description:

Get a motor controller's brake state.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest.

BrakeState

Upon return, this will be the current state of the motor controller brake. Valid values are `eioON` and `eioOFF`.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioMcGetCurrent

Syntax:

```
int eioMcGetCurrent( eioTHandle Handle, int Controller,
  double *Current )
```

Description:

Use this function to read the amount of current being used by a motor.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest.

Current

The amount of current flowing through the motor. This is a count from 0 to 255, with 0 signifying no current and 255 signifying the maximum current.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioMcSetCurrentThreshold

Syntax:

```
int eioMcSetCurrentThreshold( eioTHandle Handle, int Controller,
  double Threshold )
```

Description:

To help prevent injury and/or damage, the motor controller features a current threshold. If a motor's current exceeds the set threshold, the motor's brake is immediately activated. Use this function to set the threshold.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest. Use `eioMC_ALL` to simultaneously set the threshold for all motors to the same value.

Threshold

The threshold will be set to this value. The threshold is a count, equivalent to that obtained using the `eioMcGetCurrent` function. Value 0 is no current and 255 is maximum current.

Return:

eioSUCCESS: No problems.

`eioERROR`: An error occurred.

eioMcGetCurrentThreshold

Syntax:

```
int eioMcGetCurrentThreshold( eioTHandle Handle, int Controller,
  double *Threshold )
```

Description:

Get a motor controller's current threshold value.

Parameters:

Handle

The handle of the Expert I/O device.

Controller

The controller of interest.

Threshold

Upon return, this will be the value of the current threshold. See the `eioMcSetCurrentThreshold` function for an explanation of how to interpret this value.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

Status

The status interface, declared in eio_poll.h, provides status information for the Expert I/O. Status messages are saved in a priority queue within the Expert I/O and delivered to the PC when requested. Status messages are prioritized based on importance. For instance, multiple digital input changes are not going to block motor current threshold violation messages.

eioPollGetStatus

Syntax:

```
int eioPollGetStatus( eioTHandle Handle, unsigned int *Status,
  unsigned int *NumStatusWaiting )
```

Description:

Read a status messages from the specified Expert I/O.

Parameters:

Handle

The handle of the Expert I/O device.

Status

Upon return, this will contain a status ID. Valid status IDs are shown below..

Status ID	Description
`eioPS_OK`	Status OK.
`eioPS_MC1_THERMAL_FLAG`	Motor controller 0 thermal flag set.
`eioPS_MC2_THERMAL_FLAG`	Motor controller 1 thermal flag set.
`eioPS_MC1_CURRENT_THRESH`	Motor controller 0 threshold current exceeded.
`eioPS_MC2_CURRENT_THRESH`	Motor controller 1 threshold current exceeded.
`eioPS_DIO_INPUT_CHANGE`	At least one digital input changed state.
`eioPS_DIO_INPUT_COMM`	A communications error occurred while accessing the digital inputs.
`eioPS_DIO_OUTPUT_COMM`	A communications error occurred while accessing the digital outputs.
`eioPS_DIO_JUMPERS_COMM`	A communications error occurred while accessing the programmable pulls.
`eioPS_EEPROM_COMM`	Error communicating with EEPROM.
`eioPS_USB_STALL`	USB endpoint stall.
`eioPS_USB_PID_ERR`	USB PID error.
`eioPS_USB_CRC5_ERR`	USB CRC5 error.
`eioPS_USB_CRC16_ERR`	USB CRC16 error.
`eioPS_USB_DFN8_ERR`	USB DFN8 error.
`eioPS_USB_BTO_ERR`	USB BTO error.
`eioPS_USB_WRT_ERR`	USB WRT error.
`eioS_USB_OWN_ERR`	USB OWN error.
`eioPS_USB_BTS_ERR`	USB BTS error.
`eioPS_USB_CABLE_DET`	USB cable has been detached.
`eioPS_GLOBAL_WDT`	Watchdog timeout.

NumStatusWaiting

Upon return, this will be the number of status messages that are waiting to be returned by the Expert I/O.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioPollGetStatusString

Syntax:

```
char *eioPollGetStatusString( unsigned int Status )
```

Description:

Get a string that describes the status returned by the `eioPollGetStatus` function.

Parameters:

Status

The status returned by `eioPollGetStatus`.

Return:

Upon return, this will point to a null terminated string that describes the status. The string must not be changed by the caller and could be modified by the next call to this function.

If a status string does not exist for the given status, NULL will be returned.

Product Information

The product information interface provides a way to query the Expert I/O for product related information. The interface is declared in `eio_prod_info.h`.

eioProdInfoGetFirmwareVersion

Syntax:

```
int eioProdInfoGetFirmwareVersion( eioTHandle Handle,
  eioTFirmwareVersion *FirmwareVersion )
```

Description:

Get the Expert I/O's firmware version number.

Parameters:

Handle

The handle of the Expert I/O device.

FirmwareVersion

Upon return, this will contain the firmware version number. `eioTFirmwareVersion` is defined as follows.

```
typedef struct
{
  unsigned short Major;
  unsigned short Minor;
  unsigned short SubMinor;
} eioTFirmwareVersion;
```

The version number is assembled as `Major.Minor.SubMinor`.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.

eioProdInfoGetFirmwareVersionEz

Syntax:

```
int eioProdInfoGetFirmwareVersionEz( eioTHandle Handle, unsigned short
*Major, unsigned short *Minor, unsigned short *SubMinor )
```

Description:

Get the Expert I/O's firmware version number. The version number is assembled as `Major.Minor.SubMinor`.

Parameters:

Handle

The handle of the Expert I/O device.

Major

Upon return, this will contain the Major portion of the version number.

Minor

Upon return, this will contain the Minor portion of the version number.

SubMinor

Upon return, this will contain the SubMinor portion of the version number.

Return:

`eioSUCCESS`: No problems.

`eioERROR`: An error occurred.