IP Win32 Driver Software
User's Manual

ACROMAG INCORPORATED
30765 South Wixom Road
P.O. BOX 437
Wixom, MI 48393-7037 U.S.A.

Tel: (248) 624-1541
Fax: (248) 624-9234

9500-330C

Contents

Introduction

IP Win32 Driver Software (IPSW-DLL-WIN) consists of low-level drivers and Windows 32 Dynamic Link Libraries (DLLs) that facilitate the development of Windows applications accessing Industry Pack modules installed on Acromag PCI Carrier Cards (e.g. APC8620) and CompactPCI Carrier Cards (e.g. AcPC8625). The software provides custom support for all Acromag Industry Pack modules (e.g. IP220) as well as general read/write access and interrupt support for IP modules from other vendors.

Note:
Most carrier control software functions are used identically with all Acromag PCI and CompactPCI Carrier Cards. The convention of this document is to refer to all these cards using the generic term "carrier."

DLL functions use the Windows _stdcall calling convention and can be accessed from a number of programming languages. In addition to the DLLs and drivers, the software includes an ActiveX control for implementing interrupt notifications in programming environments that do not support the use of callback functions. Several example C, Visual Basic and LabVIEW applications are also provided with source code.

This document covers general information on software installation, programming concepts, application development and redistribution issues. The following documents are also included in the IP Win32 Driver Software documentation set:

- Function Reference document for the IP Carrier DLL
- Function Reference documents for each Acromag IP module DLL.
- Function Reference document for the "IP Generic" DLL (which can be used to access IPs from other vendors).
- Windows XP Embedded Application Note

After reviewing this user's manual, readers will next want to consult the Function Reference documents specific to their hardware.

Hardware Support

The list of supported Acromag Industry Pack carriers and modules is shown in Table1.

Table 1: Acromag IP Carriers and Modules

Model	Description	Interrupts
AcPC8625	4 slot non-intelligent CompactPCI bus carrier board	N.A.
AcPC8630	2 slot non-intelligent CompactPCI bus carrier board	N.A.
AcPC8635	2 slot, rear I/O, non-intelligent CompactPCI bus carrier board	N.A.
APC8620	5 slot non-intelligent PCI bus carrier board	N.A.
APC8621	3 slot non-intelligent PCI bus carrier board	N.A.
APC8620A	5 slot, 32MHz capable non-intelligent PCI bus carrier board	N.A.
APC8621A	3 slot, 32MHz capable non-intelligent PCI bus carrier board	N.A.
IP-1K100	Reconfigurable FPGA module	Yes
IP-1K110	Reconfigurable FPGA module	Yes
IP-1K125	JTAG-Reconfigurable FPGA module	Yes
IP220(A)-X	8/16 Non-Isolated 12-Bit DAC Outputs	No
IP230-X	4/8 Ch., 16-Bit DAC Outputs (No RAM or interrupt)	No
IP231-X	8/16 Non-Isolated 16-Bit DAC Outputs	No
IP235-X	4/8 Ch., 16-Bit DAC Outputs with RAM and interrupt	Yes
IP236-X	4/8 Ch., 16-Bit DAC Outputs with FIFO and interrupt	Yes
IP320	40SE/20DE Non-Iso. 12-Bit ADC Inputs	No
IP330	16 Bit (16DE/32SE) Analog Input Module	Yes
IP340/1	12/14 Bit (16DE) Simultaneous Analog Input Module	Yes
IP400	40 Non-Iso. Digital Inputs	Yes
IP405	40 Non-Iso. Low Side Digital Output Switches	No
IP408	32 Non-Iso. Digital Inputs/Outputs (Low Side Sw.)	Yes
IP409	24 Non-Iso. Digital Inputs/Outputs (Differential)	Yes
IP440-X	32 Ch., Isolated Digital Input Module with Interrupts	Yes
IP445	32 Ch., Isolated SSR Output Module	No
IP470	48 Ch., Digital I/O Module with Interrupts	Yes
IP480-X	2/6, 16-Bit Counter/Timer Mod. with Int. & Watchdog	Yes
IP482	10 16-Bit Counters - TTL	Yes
IP483	5 16-Bit Counters – TTL, 2 16-Bit Counters – RS422	Yes
IP484	5 16-Bit Counters –RS422	Yes
IP500	4 Port, Serial 232 Communication	Yes
IP501	4 Port, Serial 422/485 Communication	Yes
IP502	4 Port, Serial 485 Communication	Yes
IP511	4 Port, Isolated Serial 422 Communication	Yes
IP512	4 Port, Isolated Serial 485 Communication	Yes
IP520	8 Port, Serial 232 Communication	Yes
IP521	8 Port, Serial 422/485 Communication	Yes

N.A. = Not Applicable

Language Support

IP Win32 Driver Software has been tested in the following development environments:

- Visual C++ 6.0, and .NET 2003
- Borland C++ Builder 5 and 6
- Visual Basic 6.0, and .NET 2003
- National Instruments LabVIEW 6i and 7

Getting Started

Hardware Installation

1. Plug the necessary I-Packs into the carrier. Make sure to configure any jumpers on the I-Packs as necessary.
2. With power off, install the carrier into an available slot on the PC. Connect any field wiring at this time.
3. Turn on the PC. If you are running Windows 98/Me/2000/XP you will receive a dialog box shortly after boot-up asking if you want to install a driver for the new device. Insert the IP Win32 Driver disk into the CD drive and answer yes. The New Hardware Wizard will copy and install the kernel mode driver.
   Notes:
   - The wizard may find two carrier INF files. These files are identical.
   - If the software has already been installed, the New Hardware Wizard can be directed to the "redist" subdirectory (see below) instead of the CD.

Software Installation

To install the IP Win32 Driver software, insert the IP Win32 Driver Software disk into the CD drive and run **Setup.exe**. Note that administrative rights are required to perform the installation on NT based systems.

INSTALLED SOFTWARE

The default installation directory is `C:\Program Files\Acromag\IPSW_API_WIN`.

Subdirectory	
c_examples	Microsoft Visual C++ and Borland C++ Builder examples
c_include	Header files
c_lib	COFF format import libraries
config_files	Example VHDL object code for reconfigurable I-Packs
docs	User's manual, DLL references, application notes
labview examples	LabVIEW 6i and 7 examples
redist	Carrier INF file, kernel drivers, DLLs and ActiveX control files
utility	IPEnum utility
vb_examples	Visual Basic 6 and .NET examples
vb_include	Visual Basic function prototype modules

IP Enumeration Utility

IP Win32 Driver Software includes a command line utility, **IPEnum.exe** that may be run to display basic information about all installed Acromag IP carriers. This information includes the carrier number, bus number, device number, vendor ID, device ID, PCIBAR0 base address, Irq, and the names of all installed IP modules. In addition, the utility indicates if each carrier supports IP memory space and 32MHz operation. The kernel driver and carrier DLL (APC86xx.dll) must be

installed to use this utility. Note that the carrier number is the value passed to the A86_Open function to open a connection to the carrier. (See the **Sequence of Operations** section below.)

Software Overview

The software includes a single Windows 32 DLL for carrier access and DLLs for each Acromag IP module. In most cases the name of the DLL matches the name of the IP module. There are a few exceptions, however, where groups of similar IP modules are supported by a single DLL. These include:

IP Modules	Shared DLL
IP1K100, IP1K110 and IP1K125	Ip1K100.dll
IP340 and IP341	Ip340.dll
IP482, IP483 and IP484	Ip482.dll
IP502 and IP512	Ip502.dll

The DLLs provide the Application Programming Interface (API) used to access the hardware. Each DLL is written in C and contains functions using the _stdcall calling convention. A DLL is loaded and linked at runtime when its functions are called by an executable application. Multiple applications can access the functions of a single copy of a DLL in memory.

In addition to the DLLs, the software also includes an ActiveX control that may be used to implement interrupt notifications in programming environments that do not support the use of callback functions.

Function Format

All IP carrier DLL functions have the following form:

```
status = A86_FunctionName(arg1, arg2, ... argn)
```

The format of IP module DLL functions is similar:

```
status = IPXXX_FunctionName(arg1, arg2, ... argn)
```

The "IPXXX" portion of the function name indicates the IP module the function is used with (e.g. IP470).

Every function returns a 16-bit status value. This value is set to 0 when a function completes successfully or to a negative error code if a problem occurred. The following **Status Codes** section describes the values that may be returned from the DLL functions.

For most functions, *arg1* is an integer "handle" used to reference a specific carrier or IP module. (See the **Sequence of Operations** section below.)

STATUS CODES

The table below summarizes the 16-bit status codes that may be returned from the DLL functions and ActiveX control methods. Please note the return code of any failing functions if you should need to contact Acromag for technical support.

Value	Mnemonic	Description
0	OK	Operation Successful
-1	E_INVALID_IPHNDL	Returned if no IP module is associated with the specified handle. Applies to most IP DLL functions.
-2	E_CARD_IN_USE	Returned by *A86_Open* if card is already open. This can occur if the carrier is in use by another application.
-3	E_NEWDEV	Returned by *A86_Open* if an error occurred creating a software instance of the device
-4	E_CONNECT	Returned by *A86_Open* if an error occurred connecting to the carrier. This will occur if the specified card number is invalid or if the kernel mode drivers are not properly installed or configured.
-5	E_MAPMEM	Returned by *A86_Open* or if an error occurred mapping the devices physical memory into the virtural address space.
-6	E_THREAD	Returned by *A86_Open* if an error occurred while creating the interrupt service routine thread.
-7	E_ISR_ENABLE	Returned by *A86_Open* if an error occurred while enabling interrupt support for the carrier.
-8	E_OUTOFHANDLES	Returned by *A86_Open* or *IPXXX_Open* if an attempt is made to have more than 255 carriers or I-Packs simultaneously open.
-9	E_BAD_PARAM	Returned by a function if it received an invalid parameter. This typically means the parameter was outside of the expected range or the function received a NULL pointer. Consult the individual function descriptions for other possible reasons for this error.
-10	E_INSUF_MEM	Returned by a function if there was insufficient memory to created a required data structure or carry out an operation.
-11	E_OCX_IN_USE	Returned by the ActiveX method *EnableIPXXXEvents* if the control is already configured for use by another PCI module
-12	E_DLL_LOAD	Returned by ActiveX methods if the IPXXX DLL can not be loaded
-13	E_CONFIG_READ	Returned by *A86_ReadConfigReg* if an error occurred while reading data from the device's PCI configuration space.
-14	E_TIMEOUT	Returned by a function if it timed out before completing.
-15	E_CONFIG_SET	Returned by a Configuration function if the current settings used by this function do not represent a valid configuration
-16	E_CALIB	Indicates an error generating or using calibration data.
-17	E_BUFFER	Indicates an error occurred accessing a user defined data buffer
-26	E_EX_DESIGN	Some DLL functions for reconfigurable Acromag IP modules will return this error if the IP is not configured with Acromag example design.
-29	E_UNSUPPORTED	Returned if the hardware does not support the function. For example not all carriers support 32MHz operation.

-30	E_CHECKSUM	Returned if a checksum mismatch is detected
-40	E_INVALID_CRHNDL	Returned if no IP carrier is associated with the specified handle. Applies to most IP carrier DLL functions
-41	E_EMPTY_SLOT	Returned by carrier and IP module functions if the specified carrier slot does not contain an I-Pack. Note that this error is also returned if a specified slot letter falls in the range C – E but exceeds the number of slots found on the model of carrier being accessed.
-42	E_SLOT_IN_USE	Returned by *IPXXX_Open* if the IP in the specified carrier slot is already open
-43	E_IP_MODEL	Returned by *IPXXX_Open* if the IP in the specified carrier slot is not a model supported by this DLL
-44	E_HANDSHAKE	Returned by serial communication functions if an expected handshake signal was not received

Sequence of Operations

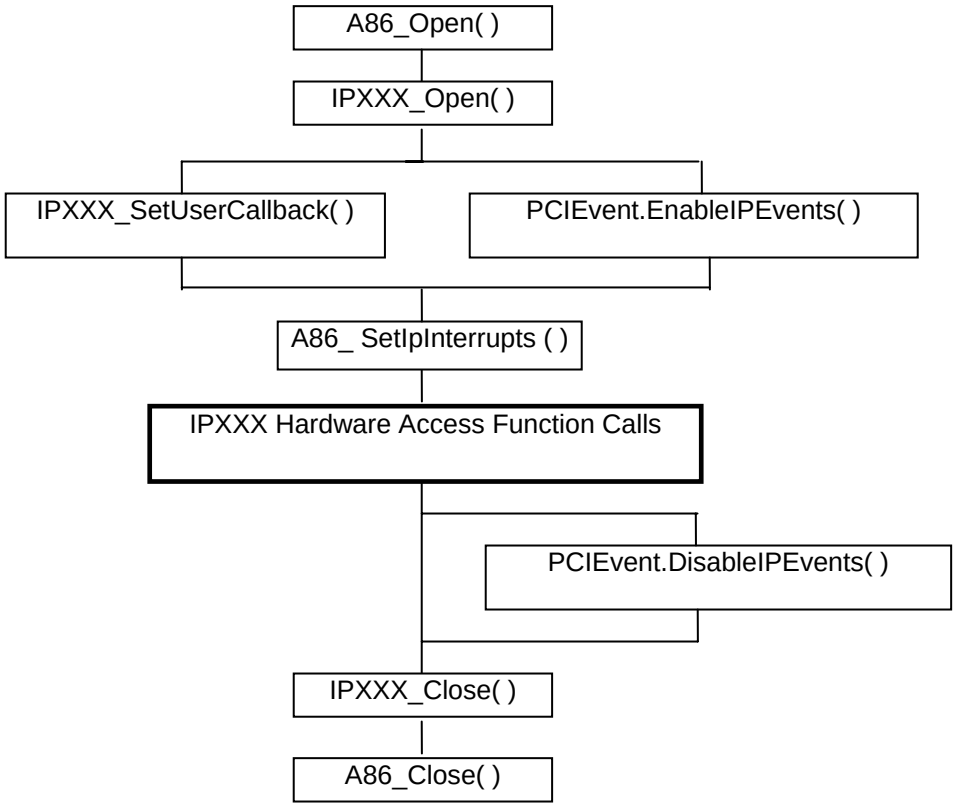
Although each IP module has its own DLL with unique functions, all IP modules are accessed in the following manner:

1. Open a connection to the carrier where the IP module is installed by calling A86_Open. This function provides an integer "carrier handle" that is used in all subsequent carrier function calls.
2. Open connections to one or more IP modules on the carrier by calling the corresponding IPXXX_Open functions. IPXXX_Open receives the carrier handle and carrier slot letter and provides an IP module handle that is used in subsequent function calls to that module.
3. (Optional) Setup a callback function or ActiveX event notification for each IP that supports interrupts. (Callback functions and event notifications are mutually exclusive.)
4. (Optional) Enable the generation of IP module interrupts by calling A86_SetIpInterrupts.
5. Implement your application logic using the carrier and IP module hardware access function calls.

Prior to terminating the application:

6. Call DisableIPEvents for each IP module using ActiveX event notifications.
7. Close each IP connection by calling IPXXX_Close.
8. Close the carrier connection by calling A86_Close.

These steps are summarized in the diagram below



Interrupts

IP Win32 Driver Software provides two mechanisms for allowing your application to respond to interrupts generated by an IP module: callback functions and ActiveX event notifications. These techniques are mutually exclusive. An application may implement one or the other for a particular module but not both. It is also acceptable for your application to not implement either option. In general, the mechanism you use will be dictated by your choice of programming language. Callback functions are the preferred technique due to lower latency, but they are not fully supported by all development environments.

Each IP DLL that supports interrupts has its own predefined internal interrupt service routine. The specifics of each routine are outlined in the IP module's corresponding Function Reference document. If you choose to implement a callback function or use ActiveX event notification, you have the option of overriding this routine. This is done by setting a "*Replace*" parameter when designating the callback function or during configuration of the ActiveX control. (See **Callback Functions** and **PCI Event ActiveX Control**)

When an interrupt occurs the following sequence of events takes place:

1. The kernel level driver disables the carrier's IP Module Interrupt Enable bit and signals the carrier DLL's internal interrupt service routine (ISR).
2. The carrier ISR identifies the IP with pending interrupts and calls the device specific ISR in the corresponding IPXXX DLL.
3. At this point four things can happen
   - If no callback or event notification was configured, the IPXXX ISR simply processes the interrupt and returns a True value to the carrier ISR.
   - If a callback function or event notification was configured but should not override the internal ISR, the internal IPXXX ISR processes the interrupt, invokes the callback or notifies the ActiveX control to fire an event and returns a True value to the carrier ISR.
   - If a callback function was configured to override the internal IPXXX ISR, the ISR invokes the callback rather than process the interrupt and then returns a True value to the carrier ISR. It is the responsibility of the callback function to process the IP interrupt.
   - If an ActiveX event notification was configured to override the internal IPXXX ISR, the ISR notifies the ActiveX control to fire an event and returns False without processing the interrupt. It is the responsibility of the event handler to process the IP interrupt and re-enable the carrier's IP Module Interrupt Enable bit.
4. The carrier ISR identifies pending IP interrupts in slot order (A – E). Steps 2 and 3 are repeated for each interrupting IP module. If all IPXXX ISRs have returned True, the carrier ISR re-enables the IP Module Interrupt Enable bit.

Note that the carrier ISR will not re-enable the carrier's IP Module Interrupt Enable bit if an interrupting IP module is using an ActiveX event handler to override the IP DLL's default ISR. This is because the event handler does not execute synchronously with the carrier ISR.

CALLBACK FUNCTIONS

Callback functions are supported in C/C++ and Visual Basic .NET.

When using the callback mechanism your application defines a function that the IP DLL will call from its internal interrupt service routine. The format of this function must exactly match that expected by the DLL. This format is hardware specific and is given in the **IPXXX_SetUserCallback** topic in the IP module's Function Reference document.

This format, however, will be some variation of the following:

```
C:   void (_stdcall *ISR)(short Handle, WORD Status)
VB7: Sub ISR(ByVal Handle As Short, ByVal Status As Short)
```

The *Handle* argument identifies the IP module that caused the interrupt. If the function is not overriding the internal ISR, the Status variable(s) will contain data allowing you to determine the cause of the interrupt (e.g. the value read from a status register by the internal ISR). If the function is overriding the internal ISR, the Status variable(s) will be zero since the internal ISR did not read any registers prior to invoking the callback function.

PCI EVENT ACTIVEX CONTROL

The PCI Event ActiveX control (**PCIEvent.ocx**) may be used for interrupt notification and processing in environments that do not support callback functions (LabVIEW) or where there are complications implementing thread safe code (versions of Visual Basic prior to Visual Basic .NET).

Note**:** Although the ActiveX control may also be used in Visual Basic .NET (VB7) and windowed C/C++ applications, the callback approach is recommended due to its lower latency.

Instructions on adding the ActiveX control to an application and defining event handlers is deferred until the Visual Basic 6 and LabVIEW topics of the **Building Windows Applications** section. The general rules for using the control, however, are as follows:

1. The PCI Event control can only be associated with one IP module at a time. Add one control to the application for each IP module you wish to receive interrupt notifications from.
2. To associate the control with a board, call the ActiveX method *EnableIPEvents* passing the IP handle received from *IPXXX_Open* and the IP's Model Code. Set the method's *Replace* parameter to indicate whether your event handler should override the DLL's internal ISR.
3. In many cases the PCI Event control can fire two types of events for the same interrupt condition. One type will pass argument(s), such as the value of a status register, which can be inspected to determine the interrupt source. The other type of event does not pass any arguments. The source of the interrupt can be determined from the event name.

   For example, if interrupt conditions are sensed on input channels 0 and 6 of an IP408, the control will fire *Bit0*, *Bit6* and *PCIEvent1w(0x41)*.

   Consult the DLL's Function Reference document to determine which events can be fired for your hardware. In general, applications will include just one of the handlers for a given interrupt condition. Which handlers you choose to implement will depend on the nature of your application.
4. If the event handler was configured to override the IP DLL's internal ISR, the handler should call *A86_SetIpInterrupts* to re-enable the carrier's IP Module Interrupt Enable bit after servicing the interrupt.
5. Call *DisableIPEvents* to disassociate the control from a board prior to calling *IPXXX_Close* to close the module.

SYNCHRONIZATION

The DLL's interrupt service routine (ISR) executes on a different thread than that of your application. Within the DLL the ISR (which includes the call to any callback function) is delimited as a device critical section. *IPXXX_StartIsrSynch* and *IPXXX_EndIsrSynch* can be used to synchronize other application threads with the ISR thread. DLLs for Acromag IP modules that do support interrupts also provide these functions to support the synchronization of hardware access by multiple threads within an application.

Bracketing a section of code between calls of *IPXXX_StartIsrSynch* and *IPXXX_EndIsrSynch* defines that code as a device critical section. Two threads within a single process cannot execute critical section code simultaneously. *IPXXX_StartIsrSynch* should be called by your application before it attempts to access data or device memory that can be accessed by another thread. Remember to call *IPXXX_EndIsrSynch* when finished accessing these shared resources.

Code in an ActiveX event handler function is not automatically defined as a critical section. If desired, *IPXXX_StartIsrSynch* and *IPXXX_EndIsrSynch* may be used to bracket this code and synchronize its execution with your application.

BASE ADDRESS POINTERS

Each IP DLL provides a function that returns the base address of the user mode mapping of the IP module's I/O space.

C and C++ programmers can cast the returned value to a byte pointer and access memory using normal pointer mechanisms. This method can be used to write additional functions that complement those provided through the DLL.

Example

```
/* Read IP408 Digital Input Channel Register B */

DWORD base_address;
volatile BYTE* pbase_addr;
WORD chan_val;

if (IP408_GetBaseAddress(Handle, &base_address) == 0)
{
   pbase_addr = (BYTE*)base_address;
   chan_val = *(PWORD)(pbase_addr + 0x2);
}
```

Building Windows Applications

This section describes the basic steps to create applications that use the IP Win32 dynamic link libraries and ActiveX control.

Steps are outlined for building applications in the following development environments:

- Microsoft Visual C++ 6 (VC6)
- Microsoft Visual C++.NET (VC7)
- Borland C++ Builder
- Visual Basic 6 (VB6)
- Visual Basic .NET (VB7)
- LabVIEW 6i and 7

C/C++

MICROSOFT VISUAL C++ 6

The steps to create a C or C++ application using VC6 are as follows:

1. Open a new or existing Visual C++ project.
2. Add the path to the necessary header files (APC86xx.h, IPXXX.h, IPErrorCodes.h) to the project's **Preprocessor | Include** directories setting. This is located under **Project | Settings | C/C++**.
3. Add the path to the import libraries (APC86xx.lib, IPXXX.lib) to the project.
   To do this:
   - Open **Project | Settings | Link**
   - Select the **Input** category and modify the **Additional Library Path** field
   - Add the import library name to the **Object/library modules** field
4. Include the windows.h, APC86xx.h, IPXXX.h and IPErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "APC86xx.h"
   #include "IP408.h"
   #include "IPErrorCodes.h"
   ```
5. Build your application

MICROSOFT VISUAL C++ .NET

The steps to create a C or C++ application using VC7 are as follows:

1. Open a new or existing Visual C++ project.
2. Add the path to the necessary header files (APC86xx.h, IPXXX.h, IPErrorCodes.h) to the project. To do this, open the project's property pages, open the **C/C++** folder, select the **General** property page and modify the **Additional Include Directories** property.
3. Add the path to the import libraries (APC86xx.lib, IPXXX.lib) to the project.
   To do this:
   - Open the project's property pages
   - Open the **Linker** folder, select the **General** property page and modify the **Additional Library Directories** property
   - Select the **Input** property page and add the import library to the **Additional Dependencies** property
4. Include the windows.h, APC86xx.h, IPXXX.h and IPErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "APC86xx.h"
   #include "IP408.h"
   #include "IPErrorCodes.h"
   ```
5. Build your application

BORLAND C++ BUILDER

The steps to create a C or C++ application using C++ Builder 5 and 6 are as follows:

1. Open a new or existing C++ Builder project.
2. Add the path to the necessary header files (APC86xx.h, IPXXX.h, IPErrorCodes.h) to the project's **Include path** setting. This is located under **Project | Options | Directories/Conditionals**.
3. The provided import libraries (APC86xx.lib, IPXXX.lib) uses Microsoft's COFF format. Since the COFF format is not compatible with Borland's OMF format it is necessary to create a compatible import library using Borland's IMPLIB utility.

   IMPLIB works like this: `IMPLIB (destination lib name) (source dll)`
   For example: `IMPLIB OMF_IP408.lib IP408.dll`

4. Add the new import library to the project by selecting **Project | Add to Project…** and browsing to the lib file.
5. Include the windows.h, APC86xx.h, IPXXX.h and IPErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "APC86xx.h"
   #include "IP408.h"
   #include "IPErrorCodes.h"
   ```
6. Build your application

Visual Basic

VISUAL BASIC 6

The steps to create an application using Visual Basic 6 are as follows:

1. Open a new or existing VB6 project.
2. Add the files containing the DLL function prototypes (APC86xx.bas, IPXXX.bas) and the error code constants (IPErrorCodes.bas) to the project. To do this, select **Project | Add Module** from the menu, click on the **Existing** tab and select the desired files.

The following steps are necessary if you will be using the ActiveX control for event notifications. Skip to step 8 if you are not using the control.

3. Add the ActiveX control to the Toolbox by selecting **Project | Components** from the menu, and checking PCIEvent Control on the **Components** tab of the property sheet. Click **OK** to close the property sheet. An icon for the control will appear on the toolbox.
4. Double-click on the PCI Events icon to add the control to your projects form.
5. Select the control on the form and note the name assigned to the control in the Properties Window (e.g. "PCIEvent1").
6. Use this name within your code to access the methods for the control
   For example:

```
Dim Status As Integer
Status = PCIEvent1.EnableIPEvents(Handle, &H3, 0)
```

7. To create an event handler, select the name of the control in the code window's Object menu, and then select the event of interest from the procedure menu. This adds a handler to your code that will be invoked each time the event occurs.
8. Run your application by clicking the **Run** button.

VISUAL BASIC .NET

The steps to create an application using Visual Basic .NET are as follows:

1. Open a new or existing VB6 project.
2. Add the files containing the DLL function prototypes (APC86xx.vb, IPXXX.vb) and the error code constants (IPErrorCodes.vb) to the project. To do this, select **Project | Add Existing Item** from the menu and select the desired files.

The following steps are necessary if you will be implementing a callback function. If you will not be using a callback skip to step 9.

In Visual Basic .NET callback functions are implemented using delegates. A delegate is a class that can hold a reference to a method and is equivalent to a type-safe function pointer or a callback function.

3. Add a new module to the project that will contain the callback function. To do this, select **Project | Add Module** from the menu, select the **Module** template and select **Open**.
4. Add an ISR subroutine to the new module. The format of the routine is hardware specific and is given in the IPXXX_SetUserCallback topic in the IP module's Function Reference document.

   This format, however, will be some variation of the following:

   ```
   Sub ISR(ByVal Handle As Short, ByVal Status As Short)

   End Sub
   ```

5. In the declares section of your form code declare a garbage collection handle:

   ```
   Dim gch As GCHandle
   ```

   (Note: If the editor indicates *`GCHandle`* is an undefined type, add `Imports System.Runtime.InteropServices` to the top of the source file.)

   The DLL will store the delegate passed to it for later use. Since the DLL is unmanaged code, it is necessary to manually prevent garbage collection of the delegate until the DLL is through with it.
6. Include the following statements prior to the call to IPXXX_SetUserCallback:

   ```
   Dim dlg As IPXXX_ISRDelegate    'defined in IPXXX.vb
   dlg = AddressOf ISR       'assign delegate to callback function
   gch = GCHandle.Alloc(dlg) 'protect the delegate from garbage
                             'collection.
   ```

7. Now notify the DLL that it should invoke the callback

   ```
   IPXXX_SetUserCallback(IpHandle, dlg, fReplace )
   ```

8. At the end of your program, remember to free the garbage collection handle:

```
IPXXX_Close(ipHandle)
A86_Close(CarrierHandle)
gch.Free()
```

9. Build your application.

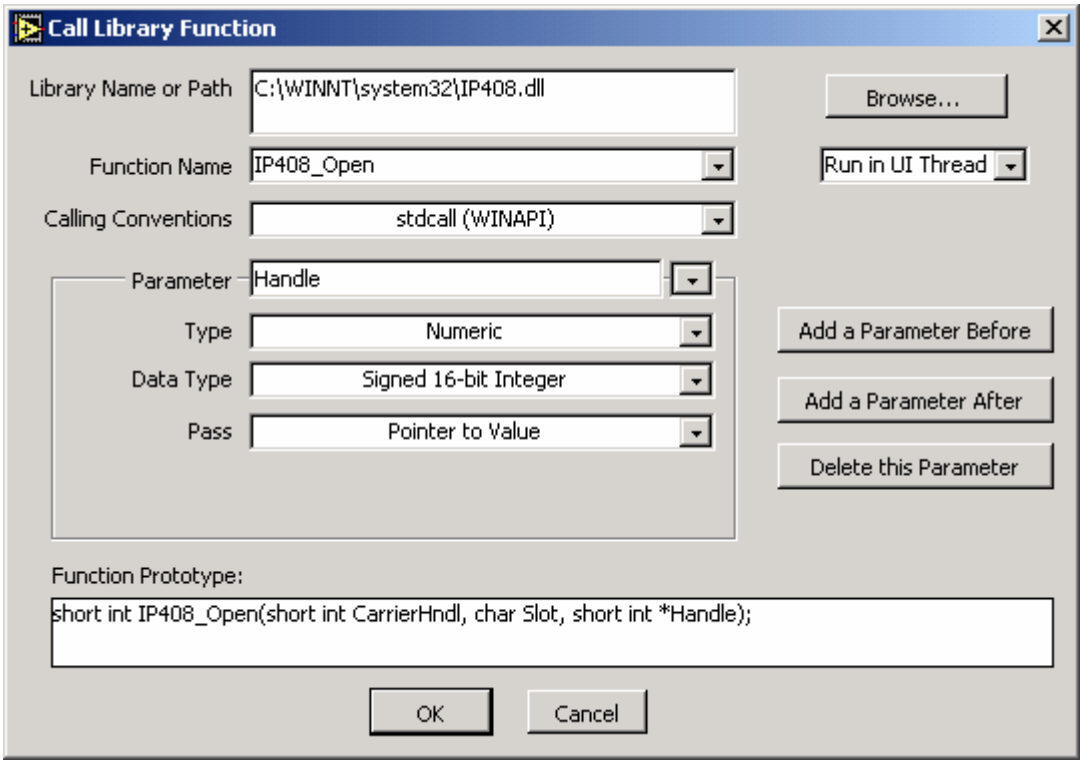
LabVIEW

The steps to create an application using LabVIEW 6i or 7 are as follows:

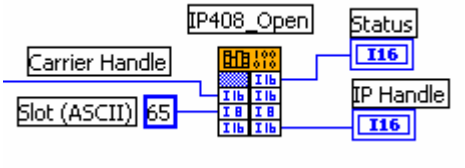
1. Open a new or existing VI.

   In LabVIEW DLL functions are called using Call Library Function nodes. The following steps outline how to add and configure one of these nodes.

2. Select **Call Library Function** from the **Functions | Advanced** subpalette and then click within the block diagram.
3. Right-click on the node and select **Configure**. This opens the Call Library Function configuration dialog.
4. Click the Browse button and locate the desired DLL within the Windows/System, WINNT/system32 or Windows/system32 directory. When finished, the name of the DLL will be displayed in the **Library Name or Path** field. Setting the Library Name in this manner automatically populates the **Function Name** Combobox.
5. Select the desired function from the **Function Name** Combobox.
6. Set the **Calling Conventions** to stdcall(WINAPI).
7. Leave the menu below the **Browse** button set to **Run in UI Thread** unless you plan on using the IsrSynch functions to make your DLL function calls thread safe.
8. Configure the return type as Numeric, Signed 16-bit Integer.
9. Add and configure additional parameters until the Function Prototype field matches the LabVIEW syntax listed for the function in the DLL's Function Reference guide. An example of a finished Call Library Function configuration dialog is shown below.



10. Wire inputs to the left side of the completed node and outputs to the right.



The following steps show how to use the PCI Event ActiveX control for event notifications in the VI.

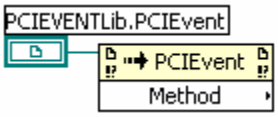
Adding the ActiveX control to a VI

1. LV6i: Select Container from the **Controls | ActiveX** subpallete and then click a point within the front panel to add the control to the panel.
   LV7: Select ActiveX Container from the **All Controls | Containers** subpallete and then click a point within the front panel to add the control to the panel.
2. Right click within the container and select the PCIEvent control. After clicking **OK** the control will appear as a white box within the front panel. In addition, a node labeled "PCIEVENTLib.PCIEvent" will appear in the block diagram window.

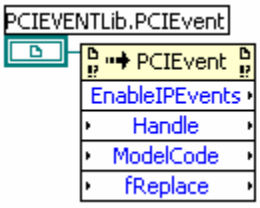
Invoking an ActiveX method

The methods, *`EnableIPEvents`* and *`DisableIPEvents`* are used to associate and disassociate the control from the handle (and hardware) used with the DLL functions. ActiveX methods are invoked as follows:

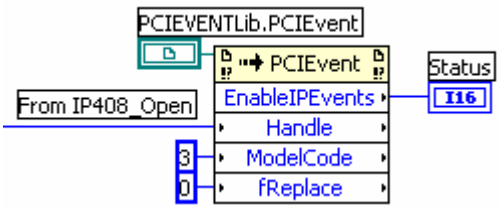
1. Select **Invoke Node** from the **Functions | Communication | ActiveX** subpalette and then click within the block diagram.
2. Wire the ActiveX control added previously to the **refnum** input on the node as shown below.



3. Right click on "Methods" to display a pop-up menu and select the desired method from the **Methods** sub-menu. The node will now show this method along with entries for its parameters.



4. Wire parameters (constants, type compatible controls and variables) to the left side of the node. Return values are wired to right of the method name.

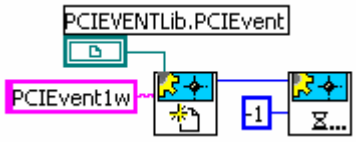


Working with ActiveX Events (LabVIEW 6i)
The PCI Event ActiveX control fires events when an interrupt occurs in the control's corresponding IP module. This section outlines how to monitor and respond to ActiveX events within the VI.

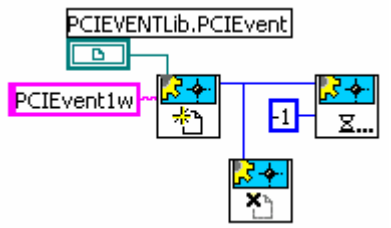
1. Select **Create ActiveX Event Queue.vi** from the **Functions | Communication | ActiveX | ActiveX Events** subpalette and then click within the block diagram.
2. Wire the previously added ActiveX control to the **refnum** input on the node.
3. Locate and right click on the **Event Name** terminal. Select **Create Constant** and type in the name of the event to monitor. This name can be obtained from the PCI module's Function Reference document. The name is <u>case sensitive</u>. The diagram should look as follows:

4. Select **Wait On ActiveX Event.vi** from the **ActiveX Events** subpalette and add the VI to the diagram. Wire the **Event Queue** terminals of the two VIs together. Locate the **ms timeout (-1)** terminal and add a constant to indicate the number of milliseconds the VI will wait for the event before timing out. A value of –1 means wait indefinitely.

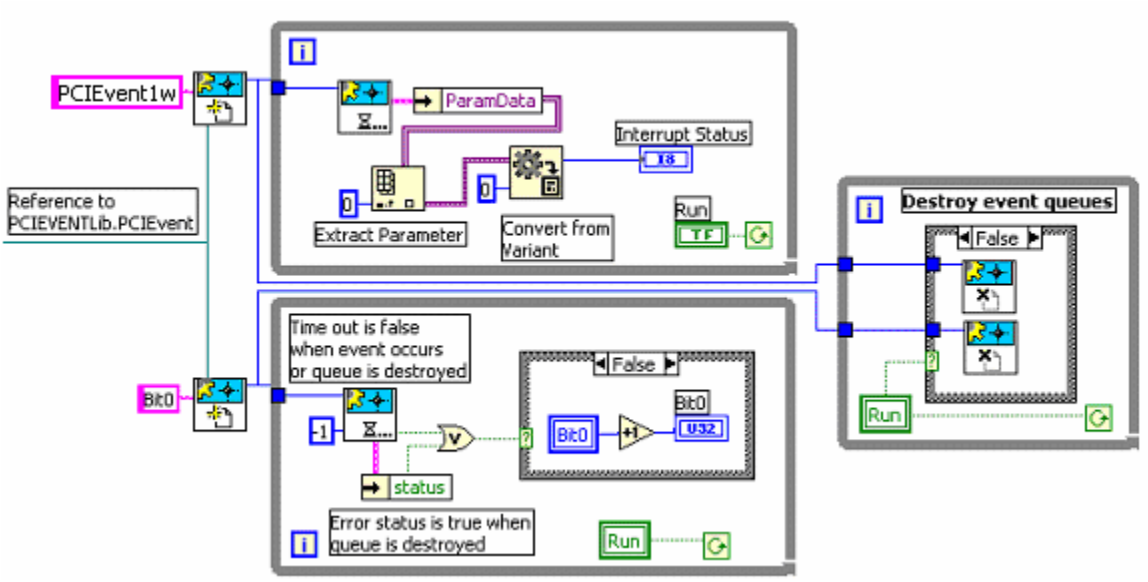
   The output wiring from the VI will depend on if the selected event passes an argument or not. If the event passes arguments, wire the Event Data terminal, otherwise, wire the timed out terminal. (See complete example at the end of this section.)



5. Select **Destroy ActiveX Event Queue.vi** for the ActiveX Events subpalette and add the VI to the diagram. Wire the VI's **Event Queue** terminal to the **Event Queue (out)** terminal of the **Wait On ActiveX Event** VI. In the completed example, a control structure is used to determine when this VI executes. (See the complete example at the end of this section.)



6. Below is a complete block diagram showing the processing of events that do (PCIEvent1w) and do not (Bit0) pass arguments.



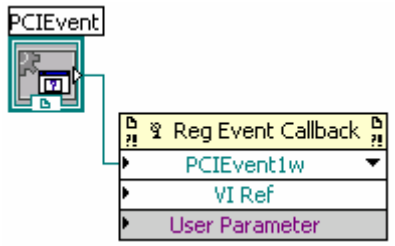
In the example above, the two Wait On ActiveX Event while loops execute as long as the Run button is activated. When the button is deactivated, both event queues are destroyed.

The upper **Wait On ActiveX Event** VI is used to monitor an event that returns a status variable. When an event occurs the VI returns the event data as a cluster. The parameter data and names are then unbundled as an array. Next, the parameter data is extracted from the array and converted from variant to integer data. Finally the status value is displayed in an indicator on the front panel.

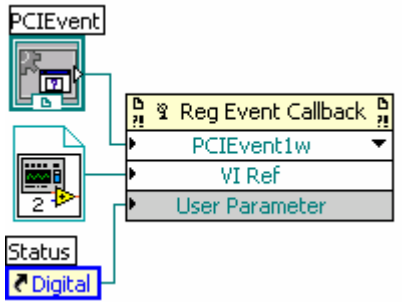
The lower **Wait On ActiveX Event** VI is used to monitor Bit0 events. The VI's **timed out** terminal returns false when an event occurs or the event queue is destroyed. If the error status is also false, a Bit0 event indicator is incremented on the front panel. (The error status is set to true when the event queue is destroyed.)

Working with ActiveX Events (LabVIEW 7)
The PCI Event ActiveX control fires events when an interrupt occurs in the control's corresponding IP module. This section outlines how to register a VI to be called when a specific ActiveX event occurs.

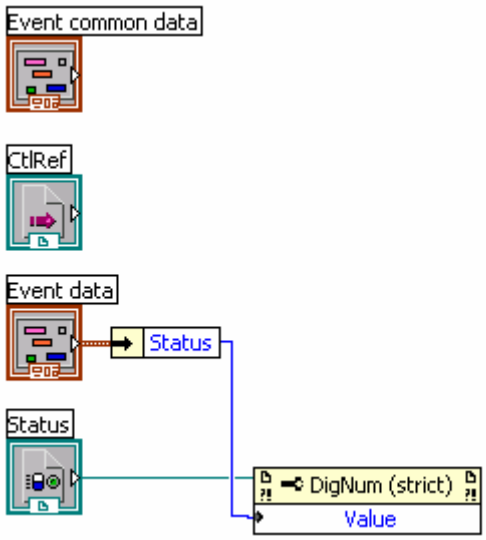
1. Select Register Event Callback from the **Functions | Communication | ActiveX** subpallete and then click within the block diagram.
2. Wire the previously added ActiveX control to the **event source ref** input on the node.
3. Click the down arrow next to the event source ref input and select the ActiveX event to be handled. The diagram should look as follows:



4. The User Parameter input can be used to transfer data between the callback VI and the main VI. For example, if a reference to an indicator on the front panel of the main VI is wired to this input, the callback VI can update the indicator each time the specified event occurs.
5. Right-click on the VI Ref portion of the node and select **Create Callback VI** from the context menu. At this point registration of the callback is complete. If necessary, the node can be resized to register callbacks for additional events.



6. The callback VI LabVIEW creates includes several nodes by default. Of primary interest are the nodes for the **User Parameter** and **Event Data**. The latter can be used to access arguments passed from the ActiveX control. Below is a complete block diagram for a callback VI that processes PCIEvent1w.



In the example above the value of the IP408's interrupt status register is passed from the ActiveX control to the Callback VI. The VI unbundles the status value from the Event data node and writes it to the Value property of the Status indicator (found on the front panel of the main VI). Note that if the callback VI handles an event that does not pass any arguments, the Event data node will not be present.

Distribution Files

IP Win32 Driver Software (IPSW-DLL-WIN) consists of three primary sets of components:

- Kernel-mode drivers
- A suite of Windows 32 Dynamic Link Libraries (DLLs)
- An event notification ActiveX control

Kernel-mode drivers

Driverx.vxd, and Drvxwdm.sys export hardware control services from the kernel. Driverx.vxd provides Windows 98 and Me support. Drvxwdm.sys provides Windows 2000 and XP support. Applications communicate indirectly with these drivers through the functions exported from the DLLs.

Windows 32 DLLs

The software includes a single Windows 32 DLL for carrier access and DLLs for each Acromag IP module. (In a few cases, groups of similar modules such as the IP482, 483, and 484 are supported by the same DLL.) These DLLs provide the Application Programming Interface (API) used to access the hardware. The carrier DLL is named "APC86xx.dll." IP DLL filenames are in the form IPXXX.dll, where "XXX" refers to the IP module's model number. Each DLL is written in C and contains functions using the _stdcall calling convention. For each DLL there is a corresponding Function Reference document that describes the functions provided by the DLL in detail.

PCI Event ActiveX Control

The PCI Event ActiveX control (PCIEvent.ocx) may be used for interrupt notification and processing in environments that do not support callback functions (LabVIEW) or where there are complications implementing thread safe code (versions of Visual Basic prior to Visual Basic .NET). In addition to the OCX file there is also a type library file (PCIEvent.tlb). Some development environments use this file to obtain information about the control's methods and events.

Redistribution Requirements

When developing an application that utilizes the driver software, the following files must be installed on the target machine.

Windows 98/Me Files

- Your application program
- The carrier DLL and all DLL's corresponding to the IP modules you are using. These are typically installed in your application's directory.
- The Microsoft® C Runtime Library (msvcr71.dll). This file is typically installed in your application's directory.
- In the \windows\system directory: Driverx.vxd

Windows 2000 Files

- Your application program
- The carrier DLL and all DLL's corresponding to the IP modules you are using. These are typically installed in your application's directory.
- The Microsoft® C Runtime Library (msvcr71.dll). This file is typically installed in your application's directory.
- In the \winnt\system32\drivers directory: Drvxwdm.sys

Windows XP Files

- Your application program
- The carrier DLL and all DLL's corresponding to the IP modules you are using. These are typically installed in your application's directory.
- The Microsoft® C Runtime Library (msvcr71.dll). This file is typically installed in your application's directory.
- In the \windows\system32\drivers directory: Drvxwdm.sys

PCI Event Control files

If you are using the PCI Event ActiveX control the following files are needed in addition to those listed above.

- PCIEvent.ocx, PCIEvent.tlb. The ActiveX control needs to be registered on the system using the Regsvr32 tool. Regsvr32.exe is included with Windows and is installed in the System (Windows 98/Me) or System32 (Windows NT) folder.
- MFCDLL Shared Library (mfc71.dll). This file is typically installed in your application's directory.

DLL Location Notes

To reduce the likelihood of "DLL Conflict" issues Microsoft recommends that DLLs be installed to the application directory with the program executable. This is the preferred location when running a single executable. However, if several applications will be simultaneously sharing a carrier or IP DLL it is recommended that the DLL be placed in a common directory. This allows the shared DLL to properly track which boards are in use.

In order for the operating system to find a DLL, its location must be part of the Windows search order. The normal search order is as follows:

1. The directory of the executable file
2. The current directory
3. The Windows system directory
4. The Windows directory
5. The directories listed in the PATH environment variable

The easiest solution to sharing a DLL is to place it in the Windows or Windows system directory. However, many applications store DLLs in these directories so using these locations creates the most risk for DLL conflict issues.

The technique used by the IPSW-API-WIN installer is to append the common DLL directory (typically C:\Program Files\Acromag\IPSW_API_WIN\redist) to the PATH environment variable. This allows the appropriate DLL to be located when running each example project.

MODIFYING THE PATH SETTING

Use the following steps if you wish to modify the PATH setting on a target machine.

Microsoft Windows 2000 and XP

1. Select Start, Settings, Control Panel, and double-click System.
2. Select the Advanced tab and then click the Environment Variables button.
3. Locate "Path" in the User Variables or System Variables. The PATH is a series of one or more directories separated by semicolons.
4. Edit the variable by appending the path to the common DLL directory to the right of the existing value.
5. Click OK

Microsoft Windows 98 and Me

1. Select Start and then click Run.
2. In the Open box, type msconfig, and then click OK.
3. In the System Configuration Utility window, click the Autoexec.bat tab (98) or Environment tab (Me).
4. Click the PATH line, click Edit, and append the path to the common DLL directory to the right of the existing value.
5. Click OK.
6. Reboot the PC for the new value to take effect.