Using the NI-VISA Driver Wizard and NI-VISA to Develop a PXI(e)/PCI(e) Driver in Windows

Publish Date: Mar 20, 2013 | 50 Ratings | **2.90** out of 5

Overview

You can use NI-VISA to program PCI(e) and PXI(e) (PCI(e) eXtensions for Instrumentation) devices installed in a PC or a PXI(e) chassis. Users writing an application for a PCI(e) or PXI(e) device with NI-VISA gain full access to the device configuration, including I/O and memory-mapped registers. NI-VISA programming is available under selected Windows OSs and the LabVIEW Real-Time Module.

Table of Contents

1. 1. Scope of This Document
2. 2. PXI(e)/PCI(e) and VISA Background
3. 3. Configuring NI-VISA to Recognize a PXI/PCI Device
4. 4. Using NI-VISA to Communicate with a PXI/PCI Device
5. 5. Using NI-VISA to Handle Events from a PXI/PCI Device
6. 6. Using LabWindows/CVI to Install Your Device .INF Files
7. 7. Summary

1. 1. Scope of This Document

This document provides an introduction to using NI-VISA and the NI-VISA Driver Wizard to develop a low-level driver for a PXI/PCI device, as well as the newer express form of devices, PXIe/PCIe. This document provides the reader with the basics of the NI-VISA features that can be used to register-level program PXI(e)/PCI(e) devices. To demonstrate the use of the VISA AP6I for this purpose, this document provides examples using a National Instruments E Series PXI data acquisition module, specifically the NI PXI-6070E. This module is used as a tool to demonstrate the features of NI-VISA; therefore, no register-level information about this module is provided beyond the example included. The only recommended methods for programming a PXI-6070E are to use the NI-DAQmx driver or the NI Measurement Hardware DDK (driver development kit).

In addition to register-level communication, this document introduces the NI-VISA event-handling model for handling interrupts from a PXI(e)/PCI(e) device. Finally, it explains how to use LabWindows/CVI to install the Windows setup files you create for your device. This document introduces the PXI(e) functionality that the NI-VISA API provides. For a full description of the NI-VISA API, refer to the *NI-VISA Help,* which should be installed on your system with the full development version of NI-VISA.

2. 2. PXI(e)/PCI(e) and VISA Background

Based on PCI, both the CompactPCI and PXI standards define a modular backplane solution packaged in a rugged mainframe topology. PXI adopts CompactPCI and extends it by adding features for integrated backplane timing and triggering, a slot-to-slot communication bus, a common software structure, and more rigid environmental standards – all essential for instrumentation systems. Because PXI is based on the CompactPCI standard, you can use PXI and CompactPCI modules in the same system without conflict. Today, PXI and CompactPCI are widely used for computer-based measurement and automation applications. PXI and CompactPCI take full advantage of Microsoft OSs, giving you an easy-to-develop, easy-to-use platform for measurement and automation. Because PXI and CompactPCI use PCI as the data communication path, PXI and CompactPCI provide a high performance measurement and automation platform.

PCI Express (PCIe) is an expansion on the older PCI standards and gives us many advantages over older bus standards, most notably higher bus throughput. Similar to PCI above, both CompactPCI Express and PXI Express (PXIe) standards define a modular backplane technology packaged in a rugged mainframe topology. PXIe similarly extends PCIe with the timing, communication and environmental standards mentioned above for PCI. PCIe, PXIe and CompactPCI Express provide the highest performance measurement and automation platform available today.

For the purpose of this tutorial we will be working with PXI/PCI devices in the NI-VISA Driver Wizard in the same manner that we would work with PXIe/PCIe devices. Therefore from this point forward, any reference to a PXI/PCI device can be considered the same as a PXIe/PCIe device.

More and more of the measurement and automation industry is adopting PXI as a standard platform. Until recently, the most common way to low-level program a PXI device was to use a Windows kernel-level driver. This process required not only extensive knowledge of the register set of the device, but also low-level knowledge of Windows programming. This development could be very time-consuming because users had to write a separate driver for each Windows OS. NI-VISA makes this process much easier by acting as the kernel-level driver for a device, thus eliminating the need for the user to develop a new Windows kernel-level driver. Using NI-VISA also eliminates the need for writing separate device drivers for each Windows OS because NI-VISA is already cross-platform compatible under Windows 7/Vista/XP/2000. The NI-VISA Driver Wizard, which is one of the tools available with the full development version of NI-VISA, even assists you in the Windows setup of your device. Through the NI-VISA Driver Wizard you can create a Windows setup (.INF) file, as well as easily set up how NI-VISA handles interrupts from your device. A detailed knowledge of the register set and register programming of your PXI/PCI device is still required, but development time is drastically reduced because NI-VISA handles the low-level Windows programming for you.

Note: Previous versions of NI-VISA included the PXI Driver Development Wizard. USB support was added to NI-VISA 3.0, and the PXI Driver Development Wizard was replaced with the NI-VISA Driver Wizard.

3. 3. Configuring NI-VISA to Recognize a PXI/PCI Device

Every PXI/PCI device must have an associated kernel-level driver. Windows uses a setup (.INF) file to associate a device and its driver. For NI-VISA to recognize your device, you must use the NI-VISA Driver Wizard to create an .INF file. The NI-VISA Driver Wizard is available from the **Start** menu under **National Instruments»VISA»Driver Wizard**.

Note: The following instructions are applicable for the NI-VISA Driver Wizard installed with NI-VISA 5.0 and above.

3.1 Hardware Bus

The NI-VISA Driver Wizard supports creating .INF files for PXI/PCI and USB devices. When the wizard opens, it displays the Hardware Bus window as in Figure 1.

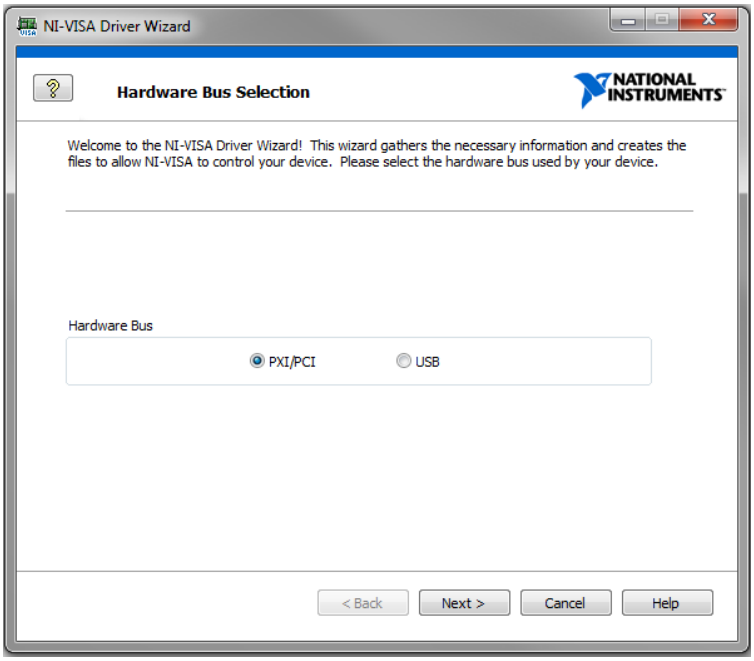


Figure 1. NI-VISA Driver Wizard Hardware Bus Selection Window

Because we want to control a PXI or a PCI device, select **PXI/PCI** and click the **Next** button. For more information about controlling a USB device, see *USB Instrument Control Tutorial* in the related links section.

3.2 Basic Device Information

The wizard will now prompt for basic information NI-VISA needs to properly locate and identify a PXI/PCI device. This basic information, such as manufacturer's identification and model code, should be documented in the register-level programming information for your PXI/PCI device, although some of it can be directly obtained from Windows Device Manager. Figure 2 shows the Basic Device Information window of the NI-VISA Driver Wizard.

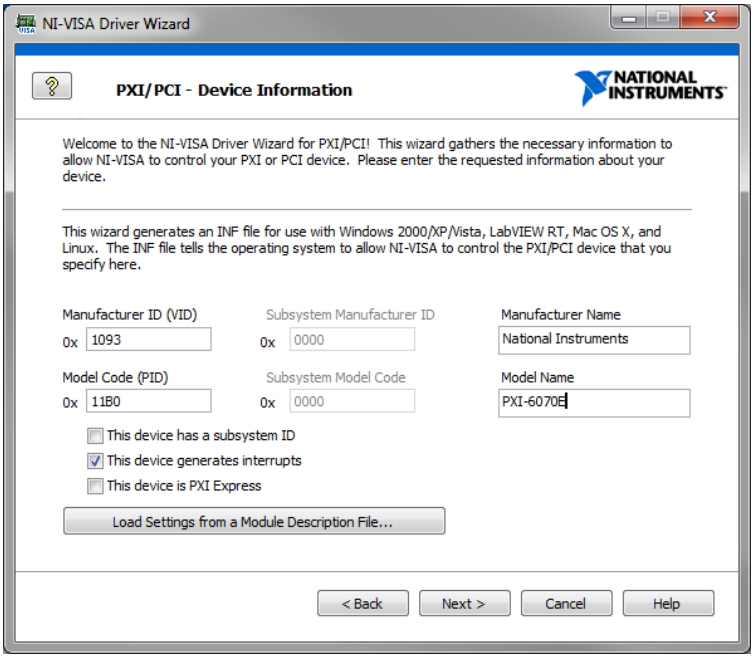


Figure 2. NI-VISA Driver Wizard PXI/PCI - Device Information

Using this window, you can specify the following basic information:

· **Manufacturer ID (VID) –** This 16-bit value identifies your device when it is installed. It is vendor specific and unique among PCI-based device providers. For example, the Manufacturer ID number for National Instruments is 0x1093.

· **Model Code (PID) –** This 16-bit value identifies your device when it is installed. It is device specific and defined by the instrument manufacturer. The model code for the PXI-6070E, which we use for our examples, is 0x11B0.

· **This device has a subsystem ID –** Some PXI/PCI devices use subsystems for identification purposes. Checking this box indicates that your device uses subsystems and that you specify the Subsystem Manufacturer ID and the Subsystem Model Code. This will also enable you to edit the Subsystem Manufacturer ID and Subsystem Model Code fields.

· **This device generates interrupts –** Some PXI/PCI devices generate interrupts to request attention from the operating system. Checking this box indicates that you need to use the VISA event-handling model in response to hardware interrupts your PXI/PCI device generates.

· **This device is PXI Express –** Checking this box indicates that you are using a PXIe or PCIe device.

· **Subsystem Manufacturer ID –** If your PXI/PCI device uses subsystems, you need to specify the Subsystem Manufacturer ID. This is a 16-bit value that identifies your PXI/PCI device when it is installed. The device manufacturer assigns the Subsystem Manufacturer ID.

· **Subsystem Model Code –** If your PXI/PCI device uses subsystems, you need to specify the Subsystem Model Code. This is a 16-bit value that identifies your device when it is installed. The device manufacturer assigns the Subsystem Model Code.

· **Manufacturer Name –** You can enter any value you would like into this field and it will not affect the performance of your device, however it is advisable to choose a descriptive name as this is how the device will be referred to in other parts of the system.

· **Model Name –** Similar to the manufacturer name, you can enter any value you would like into this field, but make sure it is descriptive to ensure ease of use.

In addition to manually entering all of the basic device and the interrupt information, the NI-VISA Driver Wizard provides the following two options to automate the process:

· **Load settings from Module Description File –** According to the PXI Specification, PXI instruments that use VISA as the low-level driver should include a module description file, also known as the module.ini file. This text file contains the information requested in the Basic Device Information window, as well as information on interrupt detection and interrupt acknowledgment. If you already have a module.ini file for your PXI device, you can load the information directly from the INI file using the **Load settings from Module Description File** button.

· **View or change loaded settings before generating files –** Once the information is loaded from the module.ini file, this information can be reviewed and edited or accepted. If you want to review the interrupt information contained in the module.ini file before generating the .INF files, select the **View or change loaded settings before generating files** checkbox. If this checkbox is not selected, you do not need to enter any more information about your PXI instrument.

Note: The **View or change loaded settings before generating files** checkbox is enabled only if the settings were loaded from a Module Description File.

To obtain the values for these settings, contact your PXI/PCI device manufacturer. Fill in these settings with that information. When all information is loaded, click the **Next** button.

The next window depends on whether you checked the **Generates interrupts** and **View or change loaded settings before generating files** boxes. If your device does not generate interrupts, you can skip to section 3.5, *Output Files Properties*. If you loaded the settings from a module description file and did not check the **View or change loaded settings before generating files** box, you can skip to section 3.6, *Installation Options*.

3.3 Interrupt Detection Information.

If the **This device generates interrupts** box is checked, the next window is the **Interrupt Detection Information** window. Because PXI/PCI devices share one of four physical interrupt lines, more than one PXI/PCI device can be interrupting at any given time. In the Interrupt Detection window, you specify the sequence of register operations so that NI-VISA can determine whether your device is interrupting. PXI/PCI hardware typically indicates a pending interrupt condition using an Interrupt Status/Control register. Figure 4 shows the Interrupt Detection Information window.

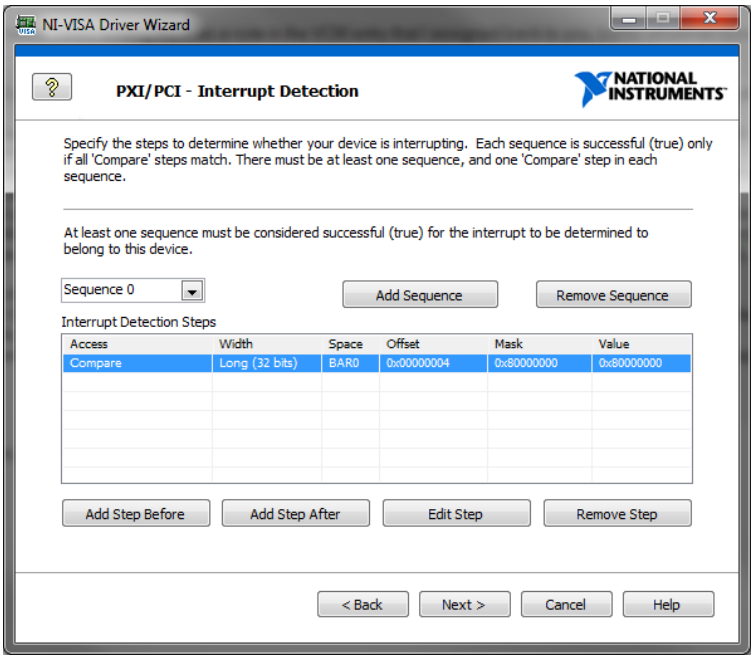


Figure 3. NI-VISA Driver Wizard PXI/PCI - Interrupt Detection Information

For the purposes of determining whether your device is asserting a hardware interrupt, a Read/Compare operation exists. This operation performs a register read, applying a user-defined mask (logical-AND) to the register contents. The resulting value is then compared with a user-specified constant (using another logical-AND). If the masked-result and the user-defined constant are the same, the comparison operation is said to be True. If the values are different, the result is False. If the result of all Read/Compare operations in a sequence of register transactions is True, NI-VISA concludes that your device is interrupting and proceeds to execute the Interrupt Acknowledge sequence. Because NI-VISA relies on the comparison operations result in making this conclusion, at least one Read/Compare operation must be present in this transaction sequence. You can add steps to the sequence using the **Add a step before** and **Add a step after** buttons. Figure 4 shows the window that appears if you click one of the **Add a step** buttons.

When determining whether your device is asserting a hardware interrupt, you can use more than one transaction sequence. All comparisons within any given detection transaction sequence must have a result of True for that detection transaction sequence to have a result of True. If multiple detection transaction sequences are present, the results from every sequence are compared using a logical-OR. If any of the sequences have a result of True, NI-VISA concludes that this interrupt belongs to this device. The steps and sequences required to detect if your PXI/PCI device is generating an interrupt should be in the register documentation for your device. You can add and remove sequences using the **Add Sequence** and **Remove Sequence** buttons, respectively.

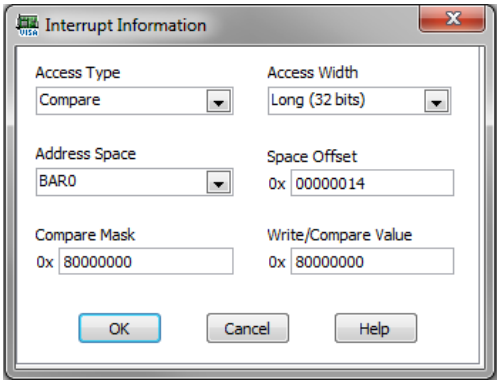


Figure 4. NI-VISA Driver Wizard Interrupt Information

In the above example shown in Figure 4, we have specified that to make this determination we must read a 32-bit value from BAR0 at offset 0x14. This value must then be checked to determine the status of bit 31 (highest order bit in the register). If bit 31 is high, NI-VISA knows that our device is generating an interrupt. Using the Compare mask, we can mask in the particular bits we need to compare, in this case bit 31. The hexadecimal value that corresponds to bit 31 being high is 0x80000000. The **Write/Compare Value** is the value that we expect the 32-bit register to be equal to after applying the mask. If we were doing a Write instead of a Read/Compare, we would use this value to specify what should be written back to the register. Again, this information is provided for example purposes only; it is not useful if attempting to handle interrupts for the PXI-6070E. Click **OK** when you finish entering a particular interrupt detection step. When you have added all steps for your device, click the **Next** button to continue with the wizard.

3.4 Interrupt Removal Information

If the **This gevice generates interrupts** box was checked, the next window is the **Interrupt Removal** information window. Once an interrupt is detected, it must be acknowledged and removed from the bus. Using this window, you can specify the steps required to acknowledge the interrupt.

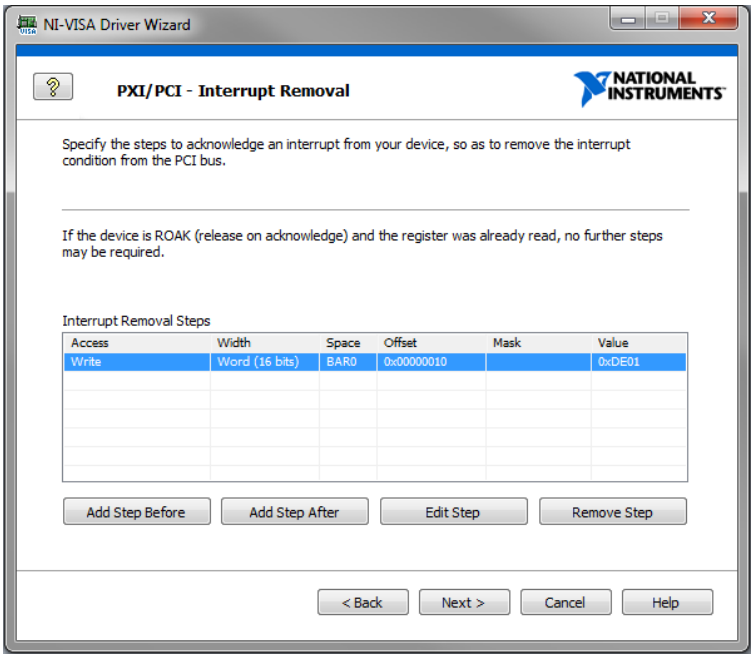


Figure 5. Interrupt Removal Information

Just as recognizing an interrupt may take multiple steps, acknowledging an interrupt may also take multiple steps. To add a step, click the **Add a step before** or **Add a step after** button. The steps to remove the interrupt from your PXI/PCI device should be in the register documentation for your device.

After selecting one of the **Add a step** buttons, the PXI Interrupt Information window appears. In Figure 6 we have specified that to remove the interrupt we must write a 16-bit value to BAR0 at offset 0x10. The **Write/Compare Value** is the value we are going to write to the 16-bit register. Click **OK** when you finish entering each particular interrupt detection step.

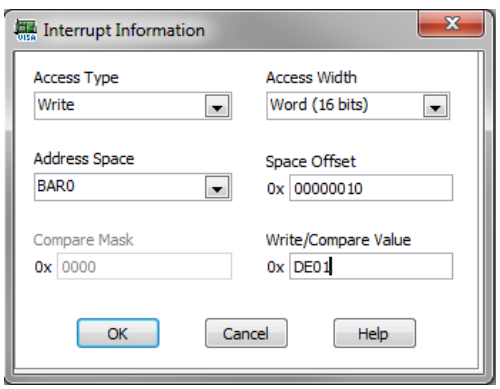


Figure 6. NI-VISA Driver Wizard Interrupt Information

There may be more than one step documented for your device. If this is the case, you continue to add steps until the sequence for your device is complete. When you have added all the steps for your device, click the **Next** button to continue with the wizard.

3.5 Output Files Properties

After you have entered all interrupt information, the Output Files Properties window appears as shown in Figure 7.

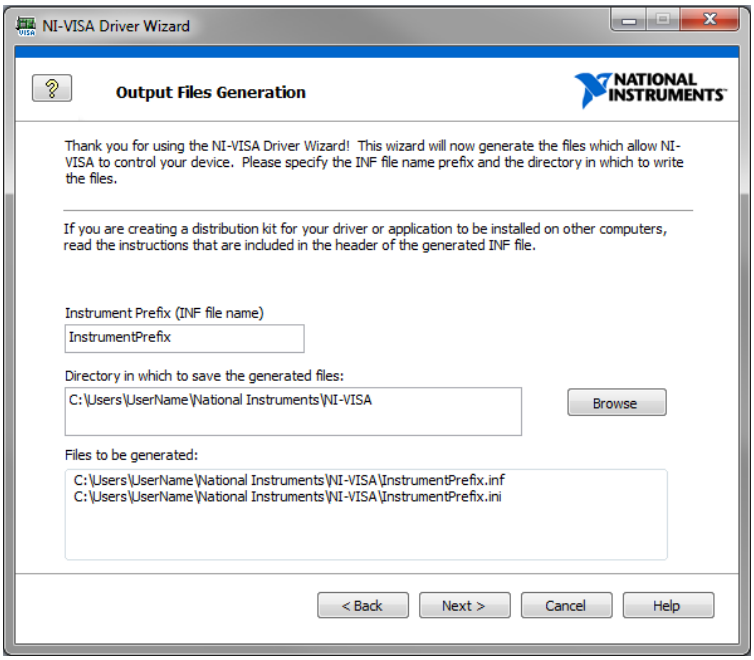


Figure 7. NI-VISA Driver Wizard Output Files Generation

The PXI Instrument Prefix is a descriptor you use to identify the files used for this device for which you can choose any value, but it will be best to choose a descriptive name. Enter a **Instrument Prefix (INF file Name)**, select the desired directory in which to place these files, and click **Next**. The .INF file is created in the directory specified by the output file directory, and the Installation Options windows appears as in Figure 8.

Along with the .INF files, the NI-VISA Driver Wizard also creates a module.ini file that you can distribute with your PXI instrument.

3.6 Installation Options

You should now have all the required Windows Setup Information (.INF) files. Before a device is visible to NI-VISA, you must use the .INF files to update the Windows system registry.

Beginning with NI-VISA 3.1, the NI-VISA Driver Wizard can automatically install the .INF file for your device. To automatically install the required files, select **Install the generated files on this computer** and click **Finish**.

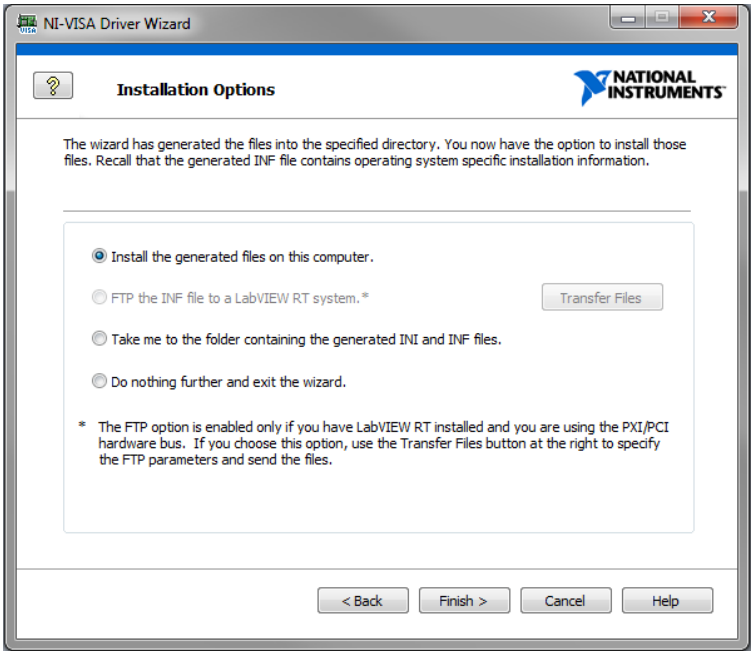


Figure 8. NI-VISA Driver Wizard Installation Options

If you would like to install the INF files yourself, you can choose either **Take me to the folder containing the generated INI and INF files** or **Do nothing and exit the wizard** and click **Finish**.

For detailed information about installing your INF file, open your INF file in a text editor, such as Notepad, and follow the instructions at the top of the file.

Note: In some cases, Windows may already have a default driver associated with your device. If this is the case, Windows may have installed that driver first. Once you've plugged in your device and Windows has installed the default driver, open the Device Manager from the Control Panel. Once Device Manager is open, expand the tree category appropriate to your device. Once you've found your device, right-click on it and choose **Update Driver Software.** If you chose to allow the NI-VISA Driver Wizard to install the drivers for you, the installer will take care of this, and no extra steps are necessary.

Once the hardware and driver is properly installed, we should now see the device through Measurement & Automation Explorer (MAX). Open MAX and press **<F5>** to refresh. Figure 9 shows what is visible in MAX once we properly install the .INF file we created using the wizard.

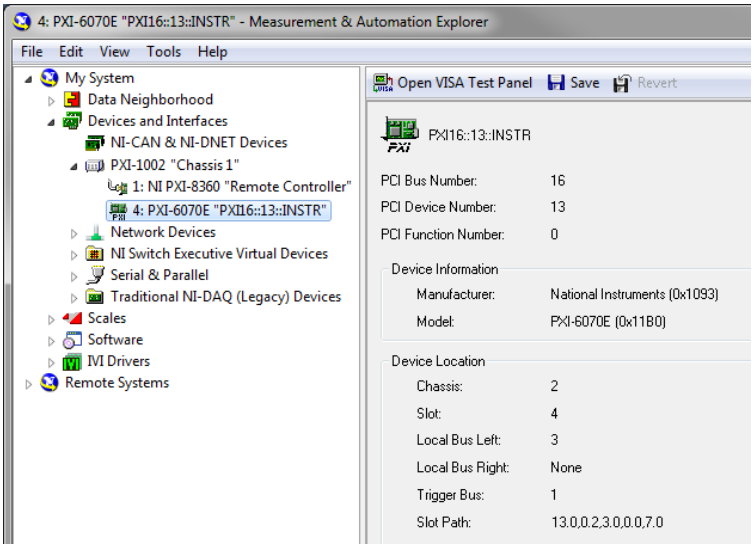


Figure 9. A Third Party PXI Device as a NI-VISA Device in MAX

Note: If you have disabled the Passport for PXI devices in VISA, you need to reenable the Passport for VISA to recognize PXI devices.

In Figure 9, notice there are several attributes specified to the right of our PXI device listing in MAX. The attributes listed under **Device Information** were set in the .INF file by the NI-VISA Driver Wizard.

Now that our PXI device is visible to VISA, it is considered a VISA PXI resource. A PXI resource is uniquely identified in the system by three characteristics:

· PCI bus number on which it is located
· PCI device number it is assigned
· Function number of the device

For single-function devices, the function number is always 0 and is optional; for multifunction devices, the function number is device specific, but is in the range of 0-7. The device number is associated with the PXI or PCI slot number, but these numbers usually differ from one PC to another. The bus number of a device is consistent from one system boot to the next, unless bridge devices are inserted somewhere between the device and the system CPU. The VISA resource descriptor you pass to the VISA function viOpen() or VISA Open in LabVIEW to access a PCI or PXI device is "PXI<bus>::<device>::<function>::INSTR". Based on the previous explanation, this can be difficult to determine until the device is installed and detected by NI-VISA in the system. As seen in Figure 9, the PXI device in this example has the resource name PXI15::14::INSTR. This name indicates that the PXI device is on PXI bus 15, is assigned PXI device number 14, and is a single function device because the function parameter is not explicitly listed.

You can determine the resource string programmatically by using the NI-VISA function viFindRsrc() or VISA Find Resource in LabVIEW. This function can determine the available NI-VISA resources in your system and return them to your program. Because these functions return all VISA resources in your system including GPIB and VXI devices, you may narrow down the VISA resources returned by these functions by supplying the viFindRsrc() or VISA Find Resource functions with a regular expression that directs it to recover only PXI resources. For example, we can use the regular expression "?*(PXI)?*" to tell the VISA driver to return only PXI device resources. We may even recover specific devices by supplying a regular expression that includes VISA attributes. For example, the regular expression "?*(PXI)?*{VI_ATTR_MANF_ID == 4243}" tells the VISA driver to return only PXI devices with a manufacturer ID of decimal value 4243. For instance, the National Instruments manufacturer ID is hexadecimal 1093, which equals decimal 4243.

Because developing the appropriate regular expression to return the specific devices you require can be very complicated, there is an easy tool in the VISA Interactive Control you can use to develop the regular expression you need. The VISA Interactive Control is installed with VISA, and can be found at **Start»Programs»National Instruments»VISA**. Once you have opened the interactive control panel, you left-click on the pull-down menu under the **Resources to Find** textbox and select **Create Query**. Here you check the resources or attributes you want to have specified in your regular expression, and it creates the regular expression for you. For more information on VISA resources, VISA resource names, and NI-VISA programming, see the *NI-VISA Help*, which should have been installed on your system with the full development version of NI-VISA.

4. 4. Using NI-VISA to Communicate with a PXI/PCI Device

Now that we can communicate with our device using NI-VISA, we will demonstrate a simple programming example to get you started programming your device using the NI-VISA API. In our example, we program the PXI-6070E to toggle a few of its digital lines. We demonstrate how to open a VISA session, map a portion of memory, peek and poke registers on the device, and close the VISA session both in LabVIEW and in LabWindows/CVI.

NI-VISA provides high-level and low-level register accesses. The valid address spaces for a PXI device are the configuration registers (VI_PXI_CFG_SPACE) and the six Base Address Registers (VI_PXI_BAR0_SPACE - VI_PXI_BAR5_SPACE). A device may use any or all of the BARs. This information is device dependent, but can be queried through the VISA attributes VI_ATTR_PXI_MEM_TYPE_BAR0-VI_ATTR_PXI_MEM_TYPE_BAR5. The values for these attributes are none (0), memory mapped (1), or I/O (2). If the value is memory mapped or I/O, you can also query the appropriate attributes for the base and size of each region. The functions in LabVIEW and LabWindows/CVI for obtaining VISA attributes are introduced shortly.

4.1 Step One – Initialize the Device

The first step in a VISA program is to open a VISA session to the device. This task is accomplished using either the viOpen() function in a text-based programming language or the VISA Open function in LabVIEW. Figure 10 shows the diagram of the LabVIEW VI we created to initialize our PXI-6070E.

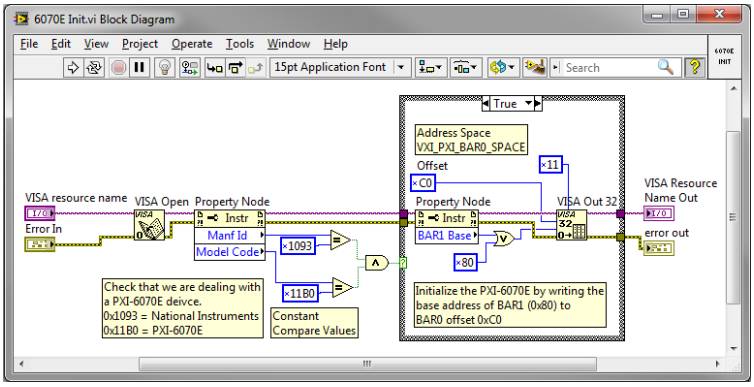


Figure 10. PXI-6070E Example - 6070E Init.VI

This VI opens a VISA session to a PXI-6070E by calling VISA Open in LabVIEW.. In the system we displayed previously in MAX, the VISA resource name "PXI15::14::INSTR" would be supplied to the **VISA Resource Name** input of VISA Open. The corresponding text-based function is viOpen(). Figure 11 shows the LabWindows/CVI code to toggle the digital lines of a PXI-6070E via the text-based version of the NI-VISA API. See the code below for the details of calling the viOpen() function in LabWindows CVI.

```
#include <ansi_c.h>
        // To program PXI devices you must define the following MACRO before
        // you include the library "visa.h"

 #if !defined NIVISA_PXI
        #define NIVISA_PXI
 #endif

        // Include the VISA library
  #include "visa.h"


int main (int argc, char *argv[])
{

                // Variable declarations
        ViUInt32 writeValue;
        ViUInt32 bar1Base;
        ViUInt16 modelCode;
        ViUInt16 manufacturerID;
        ViSession pxi6070E;
        ViStatus status;
        ViSession defaultRM;
        ViAddr address;
        ViAddr addressOffset;
        int offset;

                // Open and Initialize the PXI-6070E
                // Open the NI-VISA driver
        status = viOpenDefaultRM (&defaultRM);

                // Open a session to the PXI-6070E using its VISA resource
                // name string "PXI1::10::INSTR"
        status = viOpen (defaultRM, "PXI1::10::INSTR", VI_NULL, VI_NULL,
                                        &pxi6070E);

                // Obtain the manufacturer's ID and models code
        status = viGetAttribute (pxi6070E, VI_ATTR_MANF_ID, &manufacturerID)
        status = viGetAttribute (pxi6070E, VI_ATTR_MODEL_CODE, &modelCode);


                // Verify we have a PXI-6070E
        if(manufacturerID != 0x1093)
                return -1;
        if(modelCode != 0x11B0)
                return -1;

                //  Steps to initialize the MITE asic. These are specific
                //  to initializing the PXI-6070E, we will not go into
                //  detail about these steps
        status = viGetAttribute (pxi6070E, VI_ATTR_PXI_MEM_BASE_BAR1,
                                                &bar1Base);
        writeValue = (bar1Base | 0x80);
        status = viOut32 (pxi6070E, 11, 0xC0, writeValue);

                //  Map the block of memory we will write to using viPoke()
                //  and obtain a pointer to this memory
        status = viMapAddress (pxi6070E, 12, 0, 0x1000, 0, VI_NULL,
                                                &address);

                //  Write the values to the registers
        viPoke16 (pxi6070E, address, 0xB);
        offset = (int)address;
        offset = offset + 0x2;
```

Figure 11. LabWindows/CVI Example

There is one key difference between the LabVIEW example and the LabWindows/CVI example. The LabWindows/CVI example calls viOpenDefaultRM. This function must be called once at the beginning of every NI-VISA-based application to open and initialize the VISA driver. The handle to the driver session opened here is passed to subsequent calls to viOpen() to open sessions to specific devices. It is not necessary to explicitly call viOpenDefaultRM in LabVIEW because LabVIEW automatically calls it whenever a VI uses VISA. The error in input in the diagram of Figure 10 is an optional input in our example. This input would be supplied if our "6070E Init.vi" was used in sequence as a subVI with other subVIs that may need to pass in an error.

VISA attributes are a common way to access information about a device or its configuration. In the above examples, VISA attributes query the manufacturer ID and the model code of a PXI device to make sure the VISA session has been opened to the correct PXI device. In LabVIEW, a Property Node reads or writes VISA attributes for a device; in a text-based language, viGetAttribute() performs the same function. See the *NI-VISA Help* for more specific information about obtaining and configuring VISA attributes.

4.2 Step 2 – Communicating with the PXI Device

The final two steps in Figure 10 are specific to the PXI-6070E initialization. A property node obtains the base address of the BAR1 register. This value is then compared with the value 0x80 using a logical OR. The result is written to BAR0 space at offset 0xC0. To do this, we call the High-Level-Register-Access function VISA Out 32, which writes a 32-bit value to the specified memory space at

the specified offset. We use the decimal value 11 as an input to this VI to specify BAR0 as the address space to write to. As shown in Figure 11, the same operations are performed using the viGetAttribute and viOut32 functions in a text-based language. There are several High-Level-Register-Access functions for writing data to registers/memory space in NI-VISA. Some of these functions are introduced below. For details on using high-level and low-level functions to access registers on a device, see the *Register-Based Communication* section in the *NI-VISA Help* for more information.

Once we have initialized and opened a session to the PXI device, register peeks and pokes turn on a few of the digital lines on the device. Figure 12 is a LabVIEW block diagram demonstrating how to use VISA to write data to the registers of a PXI device. This LabVIEW example turns on some of the digital lines on the PXI-6070E. Because the values of the registers in the example are specific to the PXI-6070E, we will not go into detail as to why the specific register accesses are used.

In this VI, the Low-Level-Register-Access operation VISA Poke writes a 16-bit value to a register. When using low-level access operations, a hardware window must be set up using VISA Map Address or viMapAddress() to obtain a pointer to access the specified address space. You must specify the address space, the offset, and the window size within this space you would like to access. In our example the address space is BAR1, the offset is 0, and the size is 0x1000. Refer to Figure 11 for a text-based example of this function.

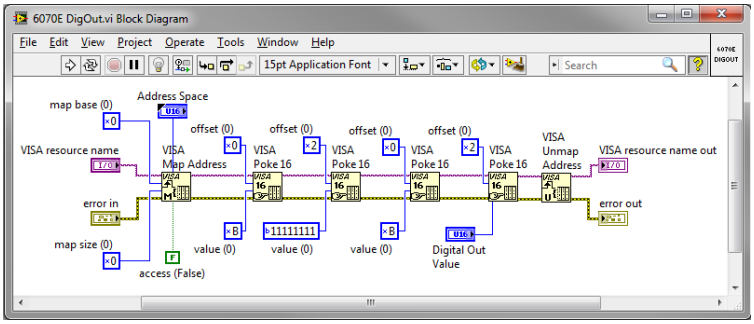
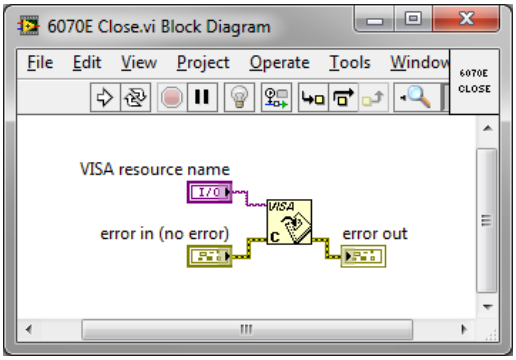


Figure 12. 6070E DigOut.vi

Once the window is properly mapped , the Poke function can write the appropriate values to the desired registers. Notice in Figure 12 that the address space is set once using VISA Map Address, but is not set in any of the VISA Poke functions. This information is passed from VISA Map Address to VISA Poke 16. The text-based example is slightly different because the pointer returned from viMapAddress() must be passed to each viPoke16 function. Notice the fourth call to VISA Poke 16 in Figure 11. Here we specify the value to be written to the eight digital lines of the PXI-6070E. The binary value that we provide to the variable **Digital Out Value** is written to these lines. In the preceding example, we have shown only a few of the Low-Level-Register-Access VISA functions. See the *Register-Based Communication* section in the *NI-VISA Help* for more information on Low-Level-Register-Access functions.

4.3 Step 3 – Closing the Device

The last step in communication is to close the VISA sessions with our PXI/PCI device and the VISA driver. Figure 13 shows a VI for closing the VISA session to the PXI-6070E.



[+] Enlarge Image

Figure 13. 6070E Close.vi

In LabVIEW, to close a session to a VISA resource that you have opened the only function you need to call is VISA Close. In the above diagram, the VISA Resource Name variable would be passed in from the previous VIs used to communicate with the device. In the LabWindows/CVI example of Figure 11, the viClose() function closes the VISA session to the PXI device. This function must actually be called twice, once to close the session to the PXI device, and once to close the session to the VISA driver. For more detailed information on closing VISA sessions, see the *NI-VISA Help*.

5. 5. Using NI-VISA to Handle Events from a PXI/PCI Device

Often, PXI/PCI devices use interrupts to request service from the system. VISA handles an interrupt from a device using its event-handling model. There are two mechanisms for handling events in the NI-VISA event-handling model – the queuing mechanism and the callback mechanism.

5.1 Queuing Mechanism

You can use the queuing mechanism in LabVIEW and LabWindows/CVI. With this technique, all occurrences of a specified event are placed in a queue. Your program can periodically poll the queue for event information or pause the program until the event has occurred and has been placed in the queue. The queuing mechanism is generally useful for noncritical events that do not need immediate attention, because you must explicitly poll for the occurrence of an event.

When using the queuing event-handling mechanism, the event queue must be polled manually to determine which events have occurred. The LabVIEW function to do this is VISA Wait on Event, and the corresponding LabWindows/CVI function is viWaitOnEvent(). When using these functions, a program waits for a specified amount of time for the event to be placed in the VISA event queue. The program pauses in this function until the event occurs or the function times out. When the specified event occurs, specific information about the event is passed back via this function.

5.2 Callback Mechanism

The callback mechanism is available in LabWindows/CVI but not in LabVIEW. This technique involves having a section of code called automatically by the VISA driver whenever a particular event occurs. The function invoked if a particular event occurs is called a callback function. The callback mechanism is useful when your application requires an immediate response. It is possible to use both queuing and callbacks in the same application.

When using callbacks, you must associate an interrupt-handling function with a particular event before you enable events using the function viEnableEvent(). The function you use to associate a handler with an event is viInstallHandler(). After calling this function and then enabling events, the function you have specified using viInstallHandler() is called asynchronously when the interrupt occurs.

5.3 Event Functions

There are a few important functions to be familiar with when using events in VISA. The first two functions enable or disable the event handling mechanism in NI-VISA. The LabVIEW function VISA Enable Event and the LabWindows/CVI function viEnableEvent() tell the VISA driver to begin waiting for a particular event. The call to enable a VISA event must specify the VISA resource to monitor for events, the type of event you want to acknowledge, and which event-handling mechanism to use. In the case of PXI/PCI devices, the type of event you enable is the VI_EVENT_PXI_INTR. This

event tells VISA to place events from that particular PXI/PCI device in the queue or to use a specified callback function when the event occurs. The LabVIEW function VISA Disable Event and the LabWindows/CVI function viDisableEvent() tell the VISA driver to stop handling the specified type of events from the specified device. For more detailed information about setting up and handling PXI/PCI interrupts, see the *Events* section in the *NI-VISA Help*.

6. 6. Using LabWindows/CVI to Install Your Device .INF Files

To run your PXI application on a target machine, you must include the .INF files generated in your PXI application installer or you must install these files manually. LabWindows/CVI is a convenient development environment for creating an installer package to redistribute VXI*plug&play* or IVI instrument drivers. LabWindows/CVI can also generate a Win32 installation package for your PXI instrument.

The **Create Distribution Kit** menu lists several options for customizing an instrument driver installation. For VISA-based PXI instrument drivers, follow these steps to create the distribution kit:

1. Generate the .INF files for your instrument using the NI-VISA Driver Wizard, as previously discussed in Section 3 2. The files created will be named simply <prefix>.inf. Maintaining these exact file names is important when using LabWindows/CVI to generate a distribution kit.

2. Open LabWindows/CVI and select **Build»Distributions»Manage Distributions**.

3. Click **New**. Choose a name and location for your new distribution. Click **OK**.

3. The Edit Installer window will pop up. Click on the **Files** tab. Under the **Local Files & Directories** section, find the INF file referenced in step 1, click on it on the right pane on the right side of the window, then choose **Add file.** Note: You can not install multiple .inf files in one directory. If you have multiple .inf files, create a separate directory for each .inf file. Do not explicitly install your file to the Windows .inf store; it will be implicitly registered and copied to the Windows .inf store during installation.

6. Once you have added any files and modified any options relevant to your distribution, click **OK**.

LabWindows/CVI generates a set of installation files (including setup.exe) for your distribution. You should redistribute all the files it creates (including the *.msi and *.cab files) to your target machine. When this installer is run on a target machine, the installation script handles the extra steps necessary to register the PXI device with NI-VISA.

You should install the your distribution and thus your device driver before you install your PXI device on your target machine. If a PXI device is installed before the .INF file, Windows marks the device as Unknown and may not properly associate the NI-VISA driver with it, or may associate a different default driver with it.

7. 7. Summary

Traditionally, the only way to program a PXI/PCI device was to write a kernel-level driver. You can use NI-VISA as the low-level kernel-level driver. This way, the programming can concentrate on the high-level functionality of the device itself. To detect, install, and communicate with a PXI/PCI device, Windows needs a .INF file. The NI VISA Driver Wizard can create and install this file automatically. Once a PXI/PCI device is properly installed using the .INF file from the NI VISA Driver Wizard, you can use the NI-VISA API to program that device. The VISA API contains functions, such as viPoke, viOut, and viMove, which you can use to access registers on a PXI/PCI device. The VISA Event Handling Model handles events. They can be handled synchronously using the queuing mechanism or asynchronously using the callback mechanism.

Related Links:

Product Manuals: NI-VISA Programmer Reference Manual
Product Manuals: NI-VISA User Manual
Developer Zone Tutorial: USB Instrument Control Tutorial