

---

# NI-VISA User Manual

---

2025-04-12



# Contents

|   |    |
|---|----|
| NI-VISA User Manual.....  | 6  |
| What is NI-VISA?.....   | 7  |
| NI-VISA New Features and Changes .....                                      | 8  |
| Getting Started with Instrument Control.....                                | 9  |
| Testing Communication with your Device using VISA Interactive Control ..... | 9  |
| Selecting an Instrument Driver Type.....                                    | 10 |
| Searching the Instrument Driver Finder .....                                | 10 |
| Searching the Instrument Driver Network (IDNet) .....                       | 11 |
| Installing LabVIEW Plug and Play Drivers .....                              | 12 |
| Installing an Interchangeable Virtual Instrument (IVI) driver.....          | 12 |
| VXIplug&play Overview .....   | 14 |
| Programming Examples .....  | 15 |
| Example of Message-Based Communication.....                                 | 15 |
| Example of Register-Based Communication .....                               | 19 |
| Example of Handling Events .....  | 22 |
| Example of Locking .....  | 26 |
| VISA Overview .....   | 31 |
| VISA Terminology.....   | 31 |
| Communication Channels: Sessions .....                                      | 33 |
| The Resource Manager .....  | 35 |
| Example of Interface Independence .....                                     | 36 |
| Initializing Your VISA Application.....                                     | 37 |
| Opening a Session .....   | 37 |
| Finding Resources .....   | 39 |
| Finding VISA Resources Using Regular Expressions .....                      | 43 |
| Attribute-Based Resource Matching .....                                     | 45 |
| Configuring a Session .....   | 47 |
| Accessing Attributes .....  | 48 |
| Common Considerations for Using Attributes.....                             | 48 |
| VISA Message-Based Communication .....                                      | 50 |
| Basic I/O Services.....   | 50 |
| Synchronous Read/Write Services .....                                       | 51 |

|  |    |
|--|----|
| Asynchronous Read/Write Services .....                       | 52 |
| Clear Service .....  | 54 |
| Trigger Service .....  | 54 |
| Status/Service Request Service .....                         | 55 |
| Example VISA Message-Based Application .....                 | 56 |
| Formatted I/O Services .....                                 | 58 |
| Formatted I/O Operations .....                               | 59 |
| I/O Buffer Operations .....                                  | 60 |
| Variable List Operations .....                               | 60 |
| Manually Flushing the Formatted I/O Buffers .....            | 61 |
| Automatically Flushing the Formatted I/O Buffers .....       | 62 |
| Resizing the Formatted I/O Buffers .....                     | 63 |
| Formatted I/O Read and Low-Level I/O Receive Buffers .....   | 63 |
| Formatted I/O Write and Low-Level I/O Transmit Buffers ..... | 64 |
| Recommendations for Using the VISA Buffers .....             | 66 |
| Formatted I/O Instrument Driver Examples .....               | 66 |
| Integers .....   | 67 |
| Floating Point Values .....                                  | 68 |
| Strings .....  | 71 |
| Data Blocks .....  | 73 |
| VISA Register-Based Communication .....                      | 76 |
| High-Level Access Operations .....                           | 77 |
| High-Level Block Operations .....                            | 79 |
| Low-Level Access Operations .....                            | 80 |
| Overview of Register Accesses from Computers .....           | 81 |
| Using VISA to Perform Low-Level Register Accesses .....      | 82 |
| Operations Versus Pointer Dereference .....                  | 83 |
| Manipulating the Pointer .....                               | 84 |
| Speed .....  | 86 |
| Ease of Use .....  | 87 |
| Accessing Multiple Address Spaces .....                      | 87 |
| Shared Memory Operations .....                               | 87 |
| VISA Events .....  | 91 |
| Supported Events .....                                       | 91 |
| Enabling and Disabling Events .....                          | 95 |
| Queuing .....  | 96 |

|   |     |
|---|-----|
| Callbacks .....   | 97  |
| Callback Modes.....   | 98  |
| Independent Queues.....   | 99  |
| The userHandle Parameter .....  | 100 |
| Queuing and Callback Mechanism Sample Code.....   | 100 |
| Event Context .....   | 104 |
| Exception Handling .....  | 105 |
| VISA Locks.....   | 108 |
| Lock Types.....   | 108 |
| Lock Sharing .....  | 108 |
| Acquiring an Exclusive Lock While Owning a Shared Lock .....  | 110 |
| Nested Locks.....   | 110 |
| Locking Sample Code .....   | 110 |
| Creating a .NET Application without Measurement Studio.....   | 115 |
| NI-VISA Utilities .....   | 116 |
| Programming GPIB Devices in VISA .....  | 120 |
| Comparison Between NI-VISA and NI-488.2 APIs .....  | 121 |
| Board-Level Programming.....  | 122 |
| Programming VXI Devices in VISA .....   | 124 |
| VXI/VME Interrupts and Asynchronous Events in VISA.....   | 125 |
| Performing Arbitrary Access to VXI Memory with VISA .....   | 126 |
| Other VXI Resource Classes and VISA .....   | 126 |
| Comparison Between NI-VISA and NI-VXI APIs .....  | 127 |
| Programming PXI Devices in NI-VISA .....  | 130 |
| User-Level Functionality.....   | 131 |
| Configuring NI-VISA to Recognize a PXI Device .....   | 131 |
| Using LabWindows/CVI to Install Your Device .inf Files.....   | 134 |
| Other PXI Resource Classes and VISA .....   | 134 |
| Using the NI-VISA Driver Wizard and NI-VISA to Register-Level Program a PXI/PCI Device under Windows..... | 135 |
| PXI and VISA Background.....  | 135 |
| Configuring NI-VISA to Recognize a PXI/PCI Device .....   | 136 |
| Hardware Bus .....  | 137 |
| Basic Device Information .....  | 137 |
| Interrupt Detection Information.....  | 139 |
| Interrupt Removal Information.....  | 141 |

|  |     |
|--|-----|
| Interrupt Disarm Information .....                         | 143 |
| PXI Express Configuration Information .....                | 145 |
| Output Files Generation .....                              | 147 |
| Installation Options .....                                 | 148 |
| Using NI-VISA to Communicate with a PXI/PCI Device .....   | 152 |
| Step 1 – Initialize the Device .....                       | 152 |
| Step 2 – Communicating with the PXI Device .....           | 155 |
| Step 3 – Closing the Device .....                          | 157 |
| Using NI-VISA to Handle Events from a PXI/PCI Device ..... | 157 |
| Programming Serial Devices in VISA .....                   | 160 |
| Default vs. Configured Communication Settings .....        | 160 |
| Controlling the Serial I/O Buffers .....                   | 162 |
| National Instruments ENET Serial Controllers .....         | 163 |
| Programming Ethernet Devices in VISA .....                 | 165 |
| VISA Sockets vs. Other Sockets APIs .....                  | 166 |
| VISA Secure Connections .....                              | 168 |
| Programming Remote Devices in NI-VISA .....                | 173 |
| How to Configure and Use Remote NI-VISA .....              | 173 |
| Programming USB Devices in VISA .....                      | 175 |
| Configuring NI-VISA to Recognize a RAW USB Device .....    | 176 |
| USB and VISA Background .....                              | 178 |
| Using NI-VISA to Communicate with Your USB Device .....    | 179 |
| Configuring NI-VISA to Control Your USB Device .....       | 181 |
| Install the .inf Files and USB Device .....                | 182 |
| Test Communication with VISA Interactive Control .....     | 183 |
| VISA Access Mechanisms .....                               | 185 |
| VISA Resource Types .....                                  | 187 |
| VISA Resource Syntax and Examples .....                    | 192 |
| NI-VISA Platform-Specific and Portability Issues .....     | 199 |
| Multiple Applications Using the NI-VISA Driver .....       | 200 |
| Low-Level Access Functions .....                           | 200 |
| Interrupt Callback Handlers .....                          | 201 |
| VXI and GPIB Platforms .....                               | 202 |
| Serial Port Support .....                                  | 203 |
| VME Support .....  | 205 |

# NI-VISA User Manual

The NI-VISA User Manual provides detailed descriptions of the product functionality and the step-by-step processes for use.

## Looking For Something Else?

For information not found in the User Manual for your product, such as specifications and API reference, browse ***Related Information***.

### Related information:

- [NI-VISA API Reference](#)
- [NI-VISA Release Notes](#)

# What is NI-VISA?

This help file describes how to use NI-VISA, the NI implementation of the VISA I/O standard, in any environment using LabWindows™/CVI™, any ANSI C compiler, or Microsoft Visual Basic. It also describes the attributes, events, and operations that comprise the VISA Application Programming Interface (API).

For information on the frameworks and programming languages NI-VISA supports, refer to the readme that installed with your version of NI-VISA. Alternatively, find the readme for a specific version of NI-VISA at [ni.com/downloads/drivers](http://ni.com/downloads/drivers).

For information on programming VISA from LabVIEW, refer to the VISA documentation included with your LabVIEW software.

For information on programming VISA from Measurement Studio, refer to the VISA documentation included with your Measurement Studio software.

## Related information:

- [NI Driver Downloads](#)
- [Using VISA in LabVIEW](#)

# NI-VISA New Features and Changes

Learn about updates—including new features and behavior changes—introduced in each version of NI-VISA.

Discover what's new in the latest releases of NI-VISA.



**Note** If you cannot find new features and changes for your version, it might not include user-facing updates. However, your version might include non-visible changes such as bug fixes and compatibility updates. For information about non-visible changes, refer to your product **Release Notes** on [ni.com](https://ni.com).

## NI-VISA 2024 Q3 New Features and Changes

- Secure network connections for Windows



# Getting Started with Instrument Control

National Instruments provides many instrument drivers to assist you in quickly getting started taking measurements with your instrument. An instrument driver is a set of VIs or functions that allow you to communicate with and control your instrument.



**Note** This tutorial is for LabVIEW on a Windows operating system.

This tutorial will guide you through the following tasks:

- Testing communication with your device using VISA Interactive Control
- Selecting an Instrument Driver Type
- Searching the Instrument Driver Finder
- Searching the Instrument Driver Network (IDNet)
- Installing your Instrument Driver

## Testing Communication with your Device using VISA Interactive Control

Complete the following steps to ensure basic communication with your instrument.

1. Launch VISA Interactive Control by navigation to **Start » All Programs » National Instruments » VISA » VISA Interactive Control**.
2. Double-click your instrument in the **Devices** tree.



**Note** If your device is not listed, ensure it is properly connected and click the **Refresh** button.

3. Click **Input/Output** at the top of the window that appears.
4. On the **Basic I/O** tab, select **\*IDN?\n** from the **Select or Enter Command** drop-down list.
5. Click **Query**.

Verify that the read operation returns the correct manufacturer and model.

## Related concepts:

- [Selecting an Instrument Driver Type](#)

# Selecting an Instrument Driver Type

In LabVIEW, you can use three different types of instrument drivers: Plug and Play, Plug & Play (project-style), and Interchangeable Virtual Instrument (IVI). This topic contains information about the available driver types.

- **Plug and Play**—A Plug and Play instrument driver is a set of functions used to control and communicate with a programmable instrument. Each function corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument.
- **Plug and Play (project-style)**—A Plug and Play (project-style) instrument driver leverages the organizational benefits of the LabVIEW project. National Instruments recommends using Plug and Play (project-style) instrument drivers for any new projects using LabVIEW 2010 or later.
- **IVI**—IVI drivers offer increased performance and flexibility for more intricate test applications that require interchangeability, state caching, and/or instrument simulation. National Instruments recommends using IVI drivers if you need interchangeability and simulation in your test applications.

## Searching the Instrument Driver Finder

Complete the following steps to locate an existing LabVIEW Plug & Play instrument driver.

1. Launch LabVIEW.
2. Open the **NI Instrument Driver Finder** by selecting **Tools » Instrumentation » Find Instrument Drivers** from the menu.
3. Double-click your instrument in the **Connected Instruments** folder in the tree view to populate the **Manufacturer** and **Additional Keywords** search fields.



**Note** If your instrument is not listed under **Connected Instruments**, click **Scan for Instruments** to detect instruments connected to your machine.

#### 4. Click **Search**.

The IDFinder displays a list of drivers available for your instrument. If IDFinder does not return any results, you may need to change your search settings or search the Instrument Driver Network (IDNet) for additional third-party instrument drivers.

### Next Steps

- If IDFinder found a driver, download and install the instrument driver.
- If IDFinder did not find a driver, search IDNet for instrument drivers.

### Related concepts:

- [Installing LabVIEW Plug and Play Drivers](#)

## Searching the Instrument Driver Network (IDNet)

If the Instrument Driver Finder did not return any results for your driver, you can check the Instrument Driver Network (IDNet) for additional third-party drivers.

Complete the following steps to find and install instrument drivers from IDNet.

1. Search the Instrument Driver Network at [ni.com/idnet](http://ni.com/idnet).
2. Enter your instrument manufacturer and model into the search field and click **Search**.
3. Click the model name in the search results to display a list of available drivers for the model. There may be multiple driver types for your instrument. For example, the Tektronix TDS1002B has both LabVIEW Plug and Play (project-style) and IVI drivers available.
4. Select your driver by clicking **Go To Driver Page**.
5. Download the driver.

### Next Steps

Download and install your driver:

- Plug and play drivers
- IVI drivers

**Related concepts:**

- [Installing an Interchangeable Virtual Instrument \(IVI\) driver](#)

**Related information:**

- [Instrument Driver Network \(IDNet\)](#)

## Installing LabVIEW Plug and Play Drivers

Complete the following steps to install a Plug and Play or a Plug and Play (project-style) driver for your instrument.



**Note** NI recommends you use Plug and Play (project-style) drivers for any new test applications you create.

1. Download the driver from the Instrument Driver Finder or the Instrument Driver Network. Ensure that you download the version that corresponds to the version of LabVIEW you are using.
2. Extract the contents of the .zip file to a subdirectory of `<labview>\instr.lib..`
3. Browse examples to get started taking measurements with your instrument.
  - (Project-style driver) Navigate to and open the .lvproj file. You can browse the examples included in the project.
  - (Other plug and play drivers) Navigate to and open the LLB that corresponds to the driver name. Browse the examples included in the LLB

## Installing an Interchangeable Virtual Instrument (IVI) driver

Complete the following steps to install an IVI driver.



**Note** In order to use the IVI driver you downloaded from IDNet, you may also need to install the driver provided with your instrument and the IVI

## Compliance Package.

1. Run the self-extracting executable.
2. Follow the prompts on the NI-IVI Specific Driver installation window to install the driver.
3. Click **Finish** when the installation completes.
4. Navigate to `<labview>/instr.lib/<driver name>` and open the LLB.

# VXIplug&play Overview

The main objective of the VXIplug&play Systems Alliance is to increase ease of use for end users through open, multi-vendor systems. The alliance members share a common vision for multivendor systems architecture, encompassing both hardware and software. This common vision enables the members to work together to define and implement standards for system-level issues.

As a step toward industry-wide software compatibility, the alliance developed one specification for I/O software—the Virtual Instrument System Architecture, or VISA. The VISA specification defines a next-generation I/O software standard not only for VXI, but also for GPIB, Serial, and other interfaces. With the VISA standard endorsed by more than 35 of the largest instrumentation companies in the industry including Tektronix, Hewlett-Packard, and National Instruments, VISA unifies the industry to make software interoperable, reusable, and able to stand the test of time. The alliance also grouped the most popular operating systems, application development environments, and programming languages into distinct frameworks and defined in-depth specifications to guarantee interoperability of components within each framework.

# Programming Examples

To help you become comfortable with VISA, the following examples avoid VISA terminology. Refer to the ***VISA Overview*** section to look at these examples again but using VISA terminology and to focus more on how they explain the VISA model.

These examples show C source code and Visual Basic syntax.

The Visual Studio C# 2015 VISA examples are in the following directory:

- `C:\Users\Public\Documents\National Instruments\NI-VISA\Examples\ .NET`

The Visual Studio C 2005 VISA examples are in the following directory:

- `C:\Users\Public\Documents\National Instruments\NI-VISA\Examples\C`

## Related concepts:

- [VISA Overview](#)

## Example of Message-Based Communication

Serial, GPIB, Ethernet, and VXI systems all have a definition of message-based communication. In GPIB, serial, and Ethernet, the messages are inherent in the design of the bus itself. For VXI, the messages actually are sent via a protocol known as word serial, which is based on register communication. In either case, the end result is sending or receiving strings.

## Related concepts:

- [Initializing Your VISA Application](#)
- [VISA Message-Based Communication](#)

## Message-Based Communication Example (C)

The following example shows the basic steps in any VISA program.



**Note** The following example uses bold text to distinguish lines of code that are different from the other introductory programming examples.

```
#include "visa.h"
#define MAX_CNT 200
int main(void)
{
    ViStatus      status;          /* For checking errors */
    ViSession      defaultRM, instr; /* Communication channels */
    ViUInt32       retCount;        /* Return count from string I/O */
    ViChar         buffer[MAX_CNT]; /* Buffer for string I/O */
    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {
        /* Error Initializing VISA...exiting */
        return -1;
    }
    /* Open communication with GPIB Device at Primary Addr 1 */
    /* NOTE: For simplicity, we will not show error checking */
    status = viOpen(defaultRM, "GPIB0::1::INSTR", VI_NULL, VI_NULL, &instr);

    /* Set the timeout for message-based communication */
    status = viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);

    /* Ask the device for identification */
    status = viWrite(instr, "*IDN?\n", 6, &retCount);
    status = viRead(instr, buffer, MAX_CNT, &retCount);

    /* Your code should process the data */

    /* Close down the system */
    status = viClose(instr);
    status = viClose(defaultRM);
    return 0;
}
```



## Message-Based Communication Example (VB)



**Note** The Visual Basic examples in this manual use the VISA data types where applicable. This feature is available only on Windows. To use this feature, select the VISA library (visa32.dll) as a reference from Visual Basic. This makes use of the type library embedded into the DLL.

```
Private Sub vbMain()
    Const MAX_CNT = 200
    Dim stat      As ViStatus
    Dim dfltRM    As ViSession
    Dim sesn      As ViSession
    Dim retCount  As Long
    Dim buffer    As String * MAX_CNT
    Rem Begin by initializing the system
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then
        Rem Error initializing VISA...exiting
        Exit Sub
    End If

    Rem Open communication with GPIB Device at Primary Addr 1
    Rem NOTE: For simplicity, we will not show error checking
    stat = viOpen(dfltRM, "GPIB0::1::INSTR", VI_NULL, VI_NULL, sesn)

    Rem Set the timeout for message-based communication
    stat = viSetAttribute(sesn, VI_ATTR_TMO_VALUE, 5000)

    Rem Ask the device for identification
    stat = viWrite(sesn, "*IDN?", 5, retCount)
    stat = viRead(sesn, buffer, MAX_CNT, retCount)

    Rem Your code should process the data

    Rem Close down the system
    stat = viClose (sesn)
    stat = viClose (dfltRM)
End Sub
```

## Message-Based Communication Example Discussion

We can break down the message-based communication example into the following steps.

1. Begin by initializing the VISA system. For this task you use `viOpenDefaultRM()`, which opens a communication channel with VISA itself. This channel has a purpose similar to a telephone line. The function call is analogous to picking up the phone and dialing the operator. From this point on, the phone line, or the value output from `viOpenDefaultRM()`, is what connects you to the VISA driver. Any communication on the line is between you and the VISA driver only. `viOpenDefaultRM()` initializes VISA and must be the first VISA function called in your program.
2. Now you must open a communication channel to the device itself using `viOpen()`. Notice that this function uses the handle returned by `viOpenDefaultRM()`, which is the variable `defaultRM` in the example, to identify the VISA driver. You then specify the address of the device you want to talk to. Continuing with the phone analogy, this is like asking the operator to dial a number for you. In this case, you want to address a GPIB device at primary address 1 on the GPIB0 bus. The value for `x` in the `GPIBx` token (`GPIB0` in this example) indicates the GPIB board to which your device is attached. This means that you can have multiple GPIB boards installed in the computer, each controlling independent buses.

The two `VI_NULL` values following the address string are not important at this time. They specify that the session should be initialized using VISA defaults. Finally, `viOpen()` returns the communication channel to the device in the parameter `instr`. From now on, whenever you want to talk to this device, you use the `instr` variable to identify it. Notice that you do not use the `defaultRM` handle again. The main use of `defaultRM` is to tell the VISA driver to open communication channels to devices. You do not use this handle again until you are ready to end the program.

3. At this point, set a timeout value for message-based communication. A timeout value is important in message-based communication to determine what should happen when the device stops communicating for a certain period of time. VISA has a common function to set values such as these: `viSetAttribute()`. This function sets values such as timeout and the termination character for the communication channel. In this example, notice that the function call to `viSetAttribute()` sets the timeout to be 5 s (5000 ms) for both reading and writing strings.
4. Now that you have the communication channel set up, you can perform string I/O using the `viWrite()` and `viRead()` functions. Notice that this is the section of the programming code that is unique for message-based communication. Opening communication channels, as described in steps 1 and 2, and closing the channels,

as described in step 5, are the same for all VISA programs. The parameters that these calls use are relatively straightforward.

- a. First you identify which device you are talking to with `instr`.
  - b. Next you give the string to send, or what buffer to put the response in.
  - c. Finally, specify the number of characters you are interested in transferring.
5. When you are finished with your device I/O, you can close the communication channel to the device with the `viClose()` function.

Notice that the program shows a second call to `viClose()`. When you are ready to shut down the program, or at least close down the VISA driver, you use `viClose()` to close the communication channel that was opened using `viOpenDefaultRM()`.

## Example of Register-Based Communication



**Note** Register-based programming applies only to VXI or PXI.

VISA has two standard methods for accessing registers. The first method uses High-Level Access functions. You can use these functions to specify the address to access; the functions then take care of the necessary details to perform the access, from mapping an I/O window to checking for failures. The drawback to using these functions is the amount of software overhead associated with them.

To reduce the overhead, VISA also has Low-Level Access functions. These functions break down the tasks done by the High-Level Access functions and let the program perform each task itself. The advantage is that you can optimize the sequence of calls based on the style of register I/O you are about to perform. However, you must be more knowledgeable about how register accesses work. In addition, you cannot check for errors easily. The following example shows how to perform register I/O using the High-Level Access functions, which is the method we recommend for new users. If you are an experienced user or understand register I/O concepts, you can use the Low-Level Access Operations.

### Related concepts:

- [Low-Level Access Operations](#)
- [VISA Register-Based Communication](#)

## Register-Based Communication Example (C)



**Note** The following example uses bold text to distinguish lines of code that are different from the other introductory programming examples.

```
#include "visa.h"
int main(void)
{
    ViStatus      status;          /* For checking errors */
    ViSession      defaultRM, instr; /* Communication channels */
    ViUInt16       deviceID;       /* To store the value */
    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status &t; VI_SUCCESS) {
        /* Error Initializing VISA...exiting */
        return -1;
    }
    /* Open communication with VXI Device at Logical Addr 16 */
    /* NOTE: For simplicity, we will not show error checking */
    status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &instr);

    /* Read the Device ID, and write to memory in A24 space */
    status = viIn16(instr, VI_A16_SPACE, 0, &deviceID);
    status = viOut16(instr, VI_A24_SPACE, 0, 0x1234);

    /* Close down the system */
    status = viClose(instr);
    status = viClose(defaultRM);
    return 0;
}
```

## Register-Based Communication Example (VB)



**Note** The Visual Basic examples in this manual use the VISA data types where applicable. This feature is available only on Windows. To use this feature, select the VISA library (`visa32.dll`) as a reference from Visual Basic. This makes use of the type library embedded into the DLL.

```
Private Sub vbMain()
    Dim stat      As ViStatus
    Dim dfltRM    As ViSession
    Dim sesn      As ViSession
```

```

    Dim deviceID    As Integer
    Rem Begin by initializing the system
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then

        Rem Error initializing VISA...exiting
        Exit Sub

    End If

    Rem Open communication with VXI Device at Logical Addr 16
    Rem NOTE: For simplicity, we will not show error checking
    stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesn)

    Rem Read the Device ID and write to memory in A24 space
    stat = viIn16(sesn, VI_A16_SPACE, 0, deviceID)
    stat = viOut16(sesn, VI_A24_SPACE, 0, &H1234)

    Rem Close down the system
    stat = viClose(sesn)
    stat = viClose(dfltRM)
End Sub

```

## Register-Based Communication Example Discussion

The general structure of the register-based communication example is very similar to that of the message-based communication example. The following are some of the basic differences:

- A different address string is used for the VXI device.
- The string functions from the message-based communication example are replaced with register functions.

The address string is still the same format as the address string in the message-based communication example, but it has replaced the GPIB with VXI. Again, remember that the difference in the address string name is the extent to which the specific interface bus will be important. Indeed, since this is a simple string, it is possible to have the program read in the string from a user input or a configuration file. Thus, the program can be compiled and is still portable to different platforms.

As you can see from the programming code, you use different functions to perform I/O with a register-based device. The functions `viIn16()` and `viOut16()` read and write 16-bit

values to registers in either the A16, A24, or A32 space of VXI. As with the message-based functions, you start by specifying which device you want to talk to by supplying the `instr` variable. You then identify the address space you are targeting, such as `VI_A16_SPACE`.

The next parameter warrants close examination. Notice that we want to read in the value of the Device ID register for the device at logical address 16. Logical addresses start at offset `0xC000` in A16 space, and each logical address gets `0x40` bytes of address space. Because the Device ID register is the first address within that `0x40` bytes, the absolute address of the Device ID register for logical address 16 is calculated as follows:

$$0xC000 + (0x40 * 16) = 0xC400$$

However, notice that the offset we supplied was 0. The reason for this is that the `instr` parameter identifies which device you are talking to, and therefore the VISA driver is able to perform the address calculation itself. The 0 indicates the first register in the `0x40` bytes of address space, or the Device ID register. The same holds true for the `viOut16()` call. Even in A24 or A32 space, although it is possible that you are talking to a device whose memory starts at `0x0`, it is more likely that the VXI Resource Manager has provided some other offset, such as `0x200000` for the memory. However, because `instr` identifies the device, and the Resource Manager has told the driver the offset address of the device's memory, you do not need to know the details of the absolute address. Just provide the offset within the memory space, and VISA does the rest. For more detailed information about other defined VXI registers, refer to the ***NI-VXI Help***.

Again, when you are done with the register I/O, use `viClose()` to shut down the system.

## Example of Handling Events

When dealing with instrument communication, it is very common for the instrument to require service from the controller when the controller is not actually looking at the device. A device can notify the controller via a service request (SRQ), interrupt, or a signal. Each of these is an asynchronous event, or simply an event. In VISA, you can handle these and other events through either callbacks or a software queue.

## Handling Events Through Callbacks

Using callbacks, you can have sections of code that are never explicitly called by the program, but instead are called by the VISA driver whenever an event occurs. Due to their asynchronous nature, callbacks can be difficult to incorporate into a traditional, sequential flow program. Therefore, we recommend the queuing method of handling events for new users.

## Handling Events Through Queuing

When using a software queue, the VISA driver detects the asynchronous event but does not alert the program to the occurrence. Instead, the driver maintains a list of events that have occurred so that the program can retrieve the information later. With this technique, the program can periodically poll the driver for event information or halt the program until the event has occurred. The following example programs an oscilloscope to capture a waveform. When the waveform is complete, the instrument generates a VXI interrupt, so the program must wait for the interrupt before trying to read the data.

### Related concepts:

- [Callbacks](#)
- [VISA Events](#)

## Handling Events Example (C)



**Note** The following example uses bold text to distinguish lines of code that are different from the other introductory programming examples.

```
#include "visa.h"
int main(void)
{
    ViStatus      status;           /* For checking errors */
    ViSession     defaultRM, instr; /* Communication channels */
    ViEvent       eventData;        /* To hold event info */
    ViUInt16      statID;           /* Interrupt Status ID */
    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {
```

```

    /* Error Initializing VISA...exiting */
    return -1;
}
/* Open communication with VXI Device at Logical Address 16 */
/* NOTE: For simplicity, we will not show error checking */
status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &instr);

/* Enable the driver to detect the interrupts */
status = viEnableEvent(instr, VI_EVENT_VXI_SIGP, VI_QUEUE, VI_NULL);

/* Send the commands to the oscilloscope to capture the */
/* waveform and interrupt when done */

status = viWaitOnEvent(instr, VI_EVENT_VXI_SIGP, 5000, VI_NULL, &eventData);
if (status < VI_SUCCESS) {

    /* No interrupts received after 5000 ms timeout */
    viClose(defaultRM);
    return -1;
}
/* Obtain the information about the event and then destroy the */
/* event. In this case, we want the status ID from the interrupt. */
status = viGetAttribute(eventData, VI_ATTR_SIGP_STATUS_ID, &statID);
status = viClose(eventData);

/* Your code should read data from the instrument and process it.*/

/* Stop listening to events */
status = viDisableEvent(instr, VI_EVENT_VXI_SIGP, VI_QUEUE);

/* Close down the system */
status = viClose(instr);
status = viClose(defaultRM);
return 0; }

```

## Handling Events Example (VB)



**Note** The Visual Basic examples in this manual use the VISA data types where applicable. This feature is available only on Windows. To use this feature, select the VISA library (visa32.dll) as a reference from Visual Basic. This makes use of the type library embedded into the DLL.

```
Private Sub vbMain()
```



```

Dim stat      As ViStatus
Dim dfltRM    As ViSession
Dim sesn      As ViSession
Dim eType     As ViEventType
Dim eData     As ViEvent
Dim statID    As Integer

Rem Begin by initializing the system
stat = viOpenDefaultRM(dfltRM)
If (stat < VI_SUCCESS) Then
    Rem Error initializing VISA...exiting
    Exit Sub
End If

Rem Open communication with VXI Device at Logical Address 16
Rem NOTE: For simplicity, we will not show error checking
stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesn)

Rem Enable the driver to detect the interrupts
stat = viEnableEvent(sesn, VI_EVENT_VXI_SIGP, VI_QUEUE, VI_NULL)

Rem Send the commands to the oscilloscope to capture the
Rem waveform and interrupt when done

stat = viWaitOnEvent(sesn, VI_EVENT_VXI_SIGP, 5000, eType, eData)
If (stat < VI_SUCCESS) Then

    Rem No interrupts received after 5000 ms timeout
    stat = viClose (dfltRM)
    Exit Sub
End If

Rem Obtain the information about the event and then destroy the
Rem event. In this case, we want the status ID from the interrupt.
stat = viGetAttribute(eData, VI_ATTR_SIGP_STATUS_ID, statID)
stat = viClose(eData)

Rem Your code should read data from the instrument and process it.

Rem Stop listening to events
stat = viDisableEvent(sesn, VI_EVENT_VXI_SIGP, VI_QUEUE)

Rem Close down the system
stat = viClose(sesn)
stat = viClose(dfltRM)
End Sub

```

## Handling Events Example Discussion

Programming with events presents some new functions to use. The first two functions you notice are `viEnableEvent()` and `viDisableEvent()`. These functions tell the VISA driver which events to listen for—in this case `VI_EVENT_VXI_SIGP`, which covers both VXI interrupts and VXI signals. In addition, these functions tell the driver how to handle events when they occur. In this example, the driver is instructed to queue (`VI_QUEUE`) the events until asked for them. Notice that `instr` is also supplied to the functions, since VISA performs event handling on a per-communication-channel basis.

Once the driver is ready to handle events, you are free to write code that will result in an event being generated. In the example above, this is shown as a comment block because the exact code depends on the device. After you have set the device up to interrupt, the program must wait for the interrupt. This is accomplished by the `viWaitOnEvent()` function. Here you specify what events you are waiting for and how long you want to wait. The program then blocks, and that thread performs no other functions, until the event occurs. Therefore, after the `viWaitOnEvent()` call returns, either it has timed out (5 s in the above example) or it has caught the interrupt. After some error checking to determine whether it was successful, you can obtain information from the event through `viGetAttribute()`. When you are finished with the event data structure (`eventData`), destroy it by calling `viClose()` on it. You can now continue with the program and retrieve the data. The rest of the program is the same as the previous examples.

Notice the difference in the way you can shut down the program if a timeout has occurred. You do not need to close the communication channel with the device, but only with the VISA driver. You can do this because when a driver channel (`defaultRM`) is closed, the VISA driver closes all I/O channels opened with it. So when you need to shut down a program quickly, as in the case of an error, you can simply close the channel to the driver and VISA handles the rest for you. However, VISA does not clean up anything not associated with VISA, such as memory you have allocated. You are still responsible for those items.

## Example of Locking

Occasionally you may need to prevent other applications from using the same resource that you are using. VISA has a service called locking that you can use to gain

exclusive access to a resource. VISA also has another locking option in which you can have multiple sessions share a lock. Because lock sharing is an advanced topic that may involve inter-process communication, see Lock Sharing for more information. The following example uses the simpler form, the exclusive lock, to prevent other VISA applications from modifying the state of the specified serial port.

### Related concepts:

- [Lock Sharing](#)
- [VISA Locks](#)

## Locking Example (C)



**Note** The following example uses bold text to distinguish lines of code that are different from the other introductory programming examples.

```
#include "visa.h"
#define MAX_CNT 200
int main(void)
{
    ViStatus      status;          /* For checking errors */
    ViSession      defaultRM, instr; /* Communication channels */
    ViUInt32       retCount;        /* Return count from string I/O */
    ViChar         buffer[MAX_CNT]; /* Buffer for string I/O */
    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {
        /* Error Initializing VISA...exiting */
        return -1;
    }
    /* Open communication with Serial Port 1 */
    /* NOTE: For simplicity, we will not show error checking */
    status = viOpen(defaultRM, "ASRL1::INSTR", VI_NULL, VI_NULL, &instr);

    /* Set the timeout for message-based communication */
    status = viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);

    /* Lock the serial port so that nothing else can use it */
    status = viLock(instr, VI_EXCLUSIVE_LOCK, 5000, VI_NULL, VI_NULL);

    /* Set serial port settings as needed */
    /* Defaults = 9600 Baud, no parity, 8 data bits, 1 stop bit */
```

```

status = viSetAttribute(instr, VI_ATTR_ASRL_BAUD, 2400);
status = viSetAttribute(instr, VI_ATTR_ASRL_DATA_BITS, 7);

/* Set this attribute for binary transfers, skip it for this text example */
/* status = viSetAttribute(instr, VI_ATTR_ASRL_END_IN, 0); */

/* Ask the device for identification */
status = viWrite(instr, "*IDN?\n", 6, &retCount);
status = viRead(instr, buffer, MAX_CNT, &retCount);

/* Unlock the serial port before ending the program */
status = viUnlock(instr);

/* Your code should process the data */

/* Close down the system */
status = viClose(instr);
status = viClose(defaultRM);
return 0;
}

```

## Locking Example (VB)



**Note** The Visual Basic examples in this manual use the VISA data types where applicable. This feature is available only on Windows. To use this feature, select the VISA library (`visa32.dll`) as a reference from Visual Basic. This makes use of the type library embedded into the DLL.

```

Private Sub vbMain()
    Const MAX_CNT = 200
    Dim stat      As ViStatus
    Dim dfltRM    As ViSession
    Dim sesn      As ViSession
    Dim retCount  As Long
    Dim buffer    As String * MAX_CNT
    Rem Begin by initializing the system
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then
        Rem Error initializing VISA...exiting
        Exit Sub
    End If

    Rem Open communication with Serial Port 1

```

```

Rem NOTE: For simplicity, we will not show error checking
stat = viOpen(dfltRM, "ASRL1::INSTR", VI_NULL, VI_NULL, sesn)

Rem Set the timeout for message-based communication
stat = viSetAttribute(sesn, VI_ATTR_TMO_VALUE, 5000)

Rem Lock the serial port so that nothing else can use it
stat = viLock(sesn, VI_EXCLUSIVE_LOCK, 5000, "", "")

Rem Set serial port settings as needed
Rem Defaults = 9600 Baud, no parity, 8 data bits, 1 stop bit
stat = viSetAttribute(sesn, VI_ATTR_ASRL_BAUD, 2400)
stat = viSetAttribute(sesn, VI_ATTR_ASRL_DATA_BITS, 7)

Rem Ask the device for identification
stat = viWrite(sesn, "*IDN?", 5, retCount)
stat = viRead(sesn, buffer, MAX_CNT, retCount)

Rem Unlock the serial port before ending the program
stat = viUnlock(sesn)

Rem Your code should process the data

Rem Close down the system
stat = viClose(sesn)
stat = viClose(dfltRM)
End Sub

```

## Locking Example Discussion

As you can see, the program does not differ with respect to controlling the instrument. The ability to lock and unlock the resource simply involves inserting the `viLock()` and `viUnlock()` operations around the code that you want to ensure is protected, as far as the instrument is concerned.

To lock a resource, you use the `viLock()` operation on the session to the resource. Notice that the second parameter is `VI_EXCLUSIVE_LOCK`. This parameter tells VISA that you want this session to be the only session that can access the device. The next parameter, 5000, is the time in milliseconds you are willing to wait for the lock. For example, another program may have locked its session to the resource before you. Using this timeout feature, you can tell your program to wait until either the other program has unlocked the session, or 5 s have passed, whichever comes first.

The final two parameters are used in the lock sharing feature of `viLock()`. For most applications, however, these parameters are set to `VI_NULL`. Notice that if the `viLock()` call succeeds, you then have exclusive access to the device. Other programs do not have access to the device at all. Therefore, you should hold a lock only for the time you need to program the device, especially if you are designing an instrument driver. Failure to do so may cause other applications to block or terminate with a failure.

When using a VISA lock over the Ethernet, the lock applies to any machine using the given resource. For example, calling `viLock()` when using a National Instruments ENET Serial controller prevents other machines from performing I/O on the given serial port.

To end the example, the application calls `viUnlock()` when it has acquired the data from the instrument. At this point, the resource is accessible from any other session in any application.

# VISA Overview

The history of instrumentation reached a milestone with the ability to communicate with an instrument from a computer. Controlling instruments programmably brought a great deal of power and flexibility with the capability to control devices faster and more accurately without the need for human supervision. Over time, application development environments such as LabVIEW and LabWindows/CVI eased the task of programming and increased productivity, but instrumentation system developers were still faced with the details of programming the instrument or the device interface bus.

Instrument programmers require a software architecture that exports the capabilities of the **devices**, not just the interface bus. In addition, the architecture needs to be consistent across the devices and interface buses. The VISA library realizes these goals. It results in a simpler model to understand, reduces the number of functions the user needs to learn, and significantly reduces the time and effort involved in programming different interfaces. Instead of using a different Application Programming Interface (API) devoted to each interface bus, you can use the VISA API whether your system uses an Ethernet, GPIB, VXI, PXI, or Serial controller.

Finally, most instruments export a specific set of commands to which they will respond. These commands are often primitive functions of the device and require several commands to group them together so that the device can perform common tasks. As a result, communicating directly with the device may require much overhead in the form of multiple commands to do task A, do task B, and so on. By driving the formation of the VXIplug&play Systems Alliance and the IVI Foundation, National Instruments has spearheaded standards for higher-level instrument drivers that use VISA. This makes it easier for the vendors of instruments to create the instrument drivers themselves, so that instrumentation system developers do not have to learn the primitive command sets of each device.

## VISA Terminology

Typical device capabilities include sending and receiving messages, responding to register accesses, requesting service, being reset, and so on. One of the underlying

premises of VISA, as defined in the previous section, is to export the capabilities of the devices—independent of the interface bus—to the user. VISA encapsulates each of these abilities into a **resource**.

A resource is simply a complete description of a particular set of capabilities of a device. For example, to be able to write to a device, you need a function you can use to send messages—`viWrite()`. In addition, there are certain details you need to consider, such as how long the function should try to communicate before timing out. Those of you familiar with this methodology might recognize this approach as object-oriented (OO) design. Indeed, VISA is based on OO design. In keeping with the terminology of OO, we call the functions of these resources **operations** and the details, such as the timeout, **attributes**.

An important VISA resource is the **INSTR Resource**. This resource encapsulates all of the basic device functions together so that you can communicate with the device through a single resource. The INSTR Resource exports the most commonly used features of these resources and is sufficient for most instrument drivers.

Other resource classes currently supported include MEMACC, INTFC, BACKPLANE, SERVANT, and SOCKET. Most of these are specific to a given hardware interface type, and are intended for advanced programmers. You can read more about these classes in VISA Resource Types.

Returning to the message-based communication example, look at the point where the program has opened a communication channel with a message-based device. Because of interface independence, it does not matter whether the device is GPIB or VXI or of any other interface type. You want to send the identification query, `*IDN?\n`, to the device. Because of the possibility that the device or interface could fail, you want to ensure that the computer will not hang in the event that the device does not receive the string. Therefore, the first step is to tell the resource to time out after 5 s (5000 ms):

```
status = viSetAttribute(instr, VI_ATTR_TMO_VALUE, 5000);
```

This sets an attribute (`VI_ATTR_TMO_VALUE`) of the resource. From this point on, all communication to this resource through this communication channel (`instr`) will have a timeout of 5 s. As you become more experienced with VISA, you will see more of the benefits of this OO approach. But for now, you can see that you can set the timeout



with an operation (function) in a manner similar to that used with other drivers. In addition, the operation you need to remember, `viSetAttribute()`, is the same operation you use to set any feature of any resource. Now you send the string to the device and read the result:

```
status = viWrite(instr, "*IDN?\n", 6, &retCount);
```

```
status = viRead(instr, buffer, MAX_CNT, &retCount);
```

This is a familiar approach to programming. You use a write operation to send a string to a device, and read the response with a read operation.

### Related concepts:

- [VISA Resource Types](#)
- [Example of Message-Based Communication](#)
- [VISA Message-Based Communication](#)

## Communication Channels: Sessions

The **Programming Examples** section used an operation called `viOpen()` to open communication channels with the instruments. In VISA terminology, this channel is known as a **session**. A session connects you to the resource you addressed in the `viOpen()` operation and keeps your communication and attribute settings unique from other sessions to the same resource. In VISA, a resource can have multiple sessions to it from the same program and for interfaces other than Serial, even from other programs simultaneously. Therefore, you must consider some things about the resource to be **local**, that is, unique to the session, and other things to be **global**, that is, common for all sessions to the resource.

If you look at the descriptions of the various attributes supported by the VISA resources, you will see that some are marked **global** (such as `VI_ATTR_INTF_TYPE`) and others are marked **local** (such as `VI_ATTR_TMO_VALUE`). For example, the interface bus that the resource is using to communicate with the device (`VI_ATTR_INTF_TYPE`) is the same for everyone using that resource and is therefore a **global attribute**. However, different programs may have different timeout

requirements, so the communication timeout value (VI\_ATTR\_TMO\_VALUE) is a **local attribute**.

Again, look at the message-based communication example. To open communication with the instrument, that is, to create a session to the INSTR Resource, you use the viOpen() operation as shown below:

```
status = viOpen(defaultRM, "GPIB0::1::INSTR", VI_NULL, VI_NULL, &instr);
```

In this case, the interface to which the instrument is connected is important, but only as a means to uniquely identify the instrument. The code above references a GPIB device on bus number 0 with primary address 1. The access mode and timeout values for viOpen() are both VI\_NULL. Other values are defined, but VI\_NULL is recommended for new users and all instrument drivers.

However, notice the statement has two sessions in the parameter list for viOpen(), defaultRM and instr. Why do you need two sessions? As you will see in a moment, viOpen() is an operation on the Resource Manager, so you must have a communication channel to this resource. However, what you want is a session to the instrument; this is what is returned in instr.

For the entire duration that you communicate with this GPIB instrument, you use the session returned in instr as the communication channel. When you are finished with the communication, you need to close the channel. This is accomplished through the viClose() operation as shown below:

```
status = viClose(instr);
```

At this point, the communication channel is closed, but you are still free to open it again or open a session to another device. Notice that you do not need to close a session to open another session. You can have as many sessions to different devices as you want.

### Related concepts:

- [Programming Examples](#)
- [Example of Message-Based Communication](#)

## The Resource Manager

What is a Resource Manager? If you have worked with VXI, you are familiar with the VXI Resource Manager. Its job is to search the VXI chassis for instruments, configure them, and then return its findings to the user. The VISA Resource Manager has a similar function. It scans the system to find all the devices connected to it through the various interface buses and then controls the access to them. Notice that the Resource Manager simply keeps track of the resources and creates sessions to them as requested. You do not go through the Resource Manager with every operation defined on a resource.

Again referring to the message-based communication example, notice that the first line of code is a function call to get a session to the Default Resource Manager:

```
status = viOpenDefaultRM(&defaultRM);
```

The `viOpenDefaultRM()` function returns a unique session to the Default Resource Manager, but does not require some other session to operate. Therefore, this function is not a part of any resource—not even the Resource Manager Resource. It is provided by the VISA driver itself and is the means by which the driver is initialized.

Now that you have a communication channel (session) to the Resource Manager, you can ask it to create sessions to instruments for you. In addition to this, VISA also defines operations that can be invoked to query the Resource Manager about other resources it knows about. You can use the `viFindRsrc()` operation to give the Resource Manager a search string, known as a regular expression, for instruments in the system. See [Initializing Your VISA Application](#) for more information about `viFindRsrc()`.

### Related concepts:

- [Communication Channels: Sessions](#)
- [Example of Message-Based Communication](#)
- [Initializing Your VISA Application](#)

## Example of Interface Independence

Now that you are more familiar with the architecture of the VISA driver, we will cover an example of how VISA provides interface independence.

Many devices available today have both a Serial port and a GPIB port. If you do not use VISA, you must learn and use two APIs to communicate with this device, depending on how you have it connected. With VISA, however, you can use a single API to communicate with this device regardless of the connection. Only the initialization code differs—for example, the resource string is different, and you may have to set the serial communication port parameters if they are different from the specified defaults. But all communication after the initialization should be identical for either bus type. Many VISA-based instrument drivers exist for these types of devices.

The existence of multi-interface devices is a trend that will continue and likely increase with the proliferation of new computer buses. This trend is also true of non-GPIB devices. Several VXI device manufacturers, for example, have repackaged their boards as PXI devices, with a similarly minimal impact on their VISA-based instrument drivers.

This example shows how VISA removes the bus details from instrument communication. The VISA library takes care of those details and allows you to program your instrument based on its capabilities.

# Initializing Your VISA Application

VISA provides a single interface for finding and accessing devices on various platforms. The VISA Resource Manager exports services that allow you to control and manage your VISA resources.

Use the VISA Resource Manager services to complete the following tasks:

- Assign unique resource addresses and resource IDs
- Locate resources
- Create sessions

Each session contains all the information necessary to configure the communication channel with a device, as well as information about the device itself. This information is encapsulated inside a generic structure called an **attribute**. You can use the attributes to configure a session or to find a particular resource.

The following steps require you to prepare your application for communication with your device.

1. [Opening a Session](#)
2. [Finding Resources](#)
3. [Configuring a Session](#)

## Opening a Session

To access any of the VISA resources, call `viOpenDefaultRM()` to get a reference to the default Resource Manager. Your application can then use the session `viOpenDefaultRM()` returns to open sessions to resources controlled by that Resource Manager, as shown in the following example.

In this example, you use the `viOpen()` call to open new sessions. In this call, you specify which resource to access by using a string that describes the resource. Refer to the **VISA Resource Syntax and Examples** section for resource string and example syntax.

Refer to the ***NI-VISA Platform-Specific and Portability Issues*** section for help in determining exactly which resource you may be accessing. In some cases, such as serial (ASRL) resources, the naming conventions with other serial naming conventions may be confusing. For example, on Windows, COM1 corresponds to ASRL1, unlike in LabVIEW, where COM1 is accessible using port number 0.

The tables in the ***VISA Resource Syntax and Examples*** section show the canonical resource name formats. NI-VISA also supports the use of aliases to make opening devices easier. On Windows, launch Measurement & Automation Explorer (MAX), choose the menu option **Tools » NI-VISA » VISA Options**, and select **Aliases** under **General Settings** to manage your aliases. On UNIX, run `visaconf` and double-click any resource to bring up a dialog box to manage the alias. NI-VISA supports alias names that include letters, numbers, and underscores. To use an alias in your program, call `viOpen()` with the alias name.

## C Example

```
#include "visa.h"
int main(void)
{
    ViStatus status;
    ViSession defaultRM, instr;

    /* Open Default RM */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error Initializing VISA...exiting */
        return -1;
    }
    /* Access other resources */
    status = viOpen(defaultRM, "GPIB::1::INSTR", VI_NULL, VI_NULL, &instr);
    /* Use device and eventually close it. */
    viClose(instr);
    viClose(defaultRM);
    return 0;
}
```

## Visual Basic Example

```
Private Sub vbMain()
```

```

    Dim stat          As ViStatus
    Dim dfltRM        As ViSession
    Dim sesn          As ViSession

    Rem Open Default RM
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then

        Rem Error initializing VISA...exiting
        Exit Sub

    End If

    Rem Access other resources
    stat = viOpen(dfltRM, "GPIB::1::INSTR", VI_NULL, VI_NULL, sesn)

    Rem Use device and eventually close it.
    stat = viClose (sesn)
    stat = viClose (dfltRM)

End Sub

```

### Related concepts:

- [VISA Resource Syntax and Examples](#)
- [NI-VISA Platform-Specific and Portability Issues](#)

## Finding Resources

You can create a session to a resource using the `viOpen()` call. However, before you use this call you need to know the exact location (address) of the resource you want to open. To find out what resources are currently available at a given point in time, you can use the search services provided by the `viFindRsrc()` operation, as shown in the following example.

As this example shows, you can use `viFindRsrc()` to get a list of matching resource names, which you can then further examine one at a time using `viFindNext()`. Remember to free the space allocated by the system by invoking `viClose()` on the list reference `fList`.

Notice that while this sample function returns a session, it does not return the

reference to the resource manager session that was also opened within the same function. In other words, there is only one output parameter, the session to the instrument itself, `instrSesn`. When your program is done using this session, it also needs to close the corresponding resource manager session. If you use this style of initialization routine, you should later get the reference to the resource manager session by querying the attribute `VI_ATTR_RM_SESSION` just before closing the INSTR session. You can then close the resource manager session with `viClose()`.

## C Example

```
#include "visa.h"

#define MANF_ID          0xFF6    /* 12-bit VXI manufacturer ID of device */
#define MODEL_CODE      0xFE     /* 12-bit or 16-bit model code of device */

/* Find the first matching device and return a session to it */
ViStatus autoConnect(ViPSession instrSesn)
{
    ViStatus      status;
    ViSession     defaultRM, instr;
    ViFindList    fList;
    ViChar        desc[VI_FIND_BUFLen];
    ViUInt32      numInstrs;
    ViUInt16      iManf, iModel;

    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error initializing VISA ... exiting */
        return status;
    }
    /* Find all VXI instruments in the system */
    status = viFindRsrc(defaultRM, "?*VXI?*INSTR", &fList, &numInstrs, desc);
    if (status < VI_SUCCESS) {

        /* Error finding resources ... exiting */
        viClose(defaultRM);
        return status;
    }
    /* Open a session to each and determine if it matches */
    while (numInstrs--) {

        status = viOpen(defaultRM, desc, VI_NULL, VI_NULL, &instr);
```



```

        if (status < VI_SUCCESS) {

            viFindNext(fList, desc);
            continue;
        }
        status = viGetAttribute(instr, VI_ATTR_MANF_ID, &iManf);
        if ((status < VI_SUCCESS) || (iManf != MANF_ID)) {

            viClose(instr);
            viFindNext(fList, desc);
            continue;
        }
        status = viGetAttribute(instr, VI_ATTR_MODEL_CODE, &iModel);
        if ((status < VI_SUCCESS) || (iModel != MODEL_CODE)) {

            viClose(instr);
            viFindNext(fList, desc);
            continue;
        }
        /* We have a match, return the session without closing it */
        *instrSesn = instr;
        viClose(fList);
        /* Do not close defaultRM, as that would close instr too */
        return VI_SUCCESS;
    }
    /* No match was found, return an error */
    viClose(fList);
    viClose(defaultRM);
    return VI_ERROR_RSRC_NFOUND;
}

```

## Visual Basic Example

```

Rem Find the first matching device and return a session to it
Private Function AutoConnect(instrSesn As ViSession) As ViStatus

    Const MANF_ID = &HFF6 '12-bit VXI manufacturer ID of a device
    Const MODEL_CODE = &H0FE '12-bit or 16-bit model code of a device

    Dim stat      As ViStatus
    Dim dfltRM    As ViSession
    Dim sesn      As ViSession
    Dim fList     As ViFindList
    Dim desc      As String * VI_FIND_BUFLen

```

```

Dim nList    As Long
Dim iManf    As Integer
Dim iModel   As Integer

stat = viOpenDefaultRM(dfltRM)
If (stat < VI_SUCCESS) Then

    Rem Error initializing VISA ... exiting
    AutoConnect = stat
    Exit Function

End If

Rem Find all VXI instruments in the system
stat = viFindRsrc(dfltRM, "?*VXI?*INSTR", fList, nList, desc)
If (stat < VI_SUCCESS) Then

    Rem Error finding resources ... exiting
    viClose (dfltRM)
    AutoConnect = stat
    Exit Function

End If

Rem Open a session to each and determine if it matches
While (nList)

    stat = viOpen(dfltRM, desc, VI_NULL, VI_NULL, sesn)
    If (stat >= VI_SUCCESS) Then

        stat = viGetAttribute(sesn, VI_ATTR_MANF_ID, iManf)
        If ((stat >= VI_SUCCESS) And (iManf = MANF_ID)) Then

            stat = viGetAttribute(sesn, VI_ATTR_MODEL_CODE, iModel)
            If ((stat >= VI_SUCCESS) And (iModel = MODEL_CODE)) Then

                Rem We have a match, return session without closing
                instrSesn = sesn
                stat = viClose (fList)
                Rem Do not close dfltRM; that would close sesn too
                AutoConnect = VI_SUCCESS
                Exit Function

            End If

        End If

    End If

End While
End If

```

```

        stat = viClose (sesn)

    End If
    stat = viFindNext(fList, desc)
    nList = nList - 1

Wend
Rem No match was found, return an error
stat = viClose (fList)
stat = viClose (dfltRM)
AutoConnect = VI_ERROR_RSRC_NFOUND

End Function

```

## Finding VISA Resources Using Regular Expressions

Using `viFindRsrc()` to locate a resource in a VISA system requires a way for you to identify which resources you are interested in. The VISA Resource Manager accomplishes this through the use of regular expressions, which specify a match for certain resources in the system. Regular expressions are strings consisting of ordinary characters as well as certain characters with special meanings that you can use to search for patterns instead of specific text. Regular expressions are based on the idea of matching, where a given string is tested to see if it **matches** the regular expression; that is, to determine if it fits the pattern of the regular expression. You can apply this same concept to a list of strings to return a subset of the list that matches the expression.

The following table defines the special characters and syntax rules used in VISA regular expressions.

| Special Characters and Operators | Meaning  |
|----------------------------------|--|
| ?                                | Matches any one character.   |
| \                                | Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (\?), it matches the ? character instead of any one character. |

| Special Characters and Operators | Meaning   |
|----------------------------------|---|
| [list]                           | Matches any one character from the enclosed list. You can use a hyphen to match a range of characters.  |
| [^list]                          | Matches any character not in the enclosed list. You can use a hyphen to match a range of characters.  |
| *                                | Matches 0 or more occurrences of the preceding character or expression.   |
| +                                | Matches 1 or more occurrences of the preceding character or expression.   |
| exp exp                          | Matches either the preceding or following expression. The OR operator   matches the entire expression that precedes or follows it and not just the character that precedes or follows it. For example, VXI GPIB means (VXI) (GPIB), not VX(I G)PIB. |
| (exp)                            | Grouping characters or expressions.   |

The priority, or precedence of the operators in regular expressions is as follows:

- The grouping operator ( ) in a regular expression has the highest precedence.
- The + and \* operators have the next highest precedence.
- The OR operator | has the lowest precedence.

The following table lists some examples of valid regular expressions that you can use with viFindRsrc().

| Regular Expression   | Sample Matches  |
|----------------------|---|
| ?*INSTR              | Matches all INSTR (device) resources.                                   |
| GPIB?*INSTR          | Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR.                         |
| GPIB[0-9]*::?*INSTR  | Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR.                         |
| GPIB[^0]::?*INSTR    | Matches GPIB1::1::1::INSTR but not GPIB0::2::INSTR or GPIB12::8::INSTR. |
| VXI?*INSTR           | Matches VXI0::1::INSTR.   |
| ?*VXI[0-9]*::?*INSTR | Matches VXI0::1::INSTR.   |
| ASRL[0-9]*::?*INSTR  | Matches ASRL1::INSTR but not VXI0::5::INSTR.                            |

| Regular Expression     | Sample Matches   |
|------------------------|--|
| ASRL1+::INSTR          | Matches ASRL1::INSTR and ASRL11::INSTR but not ASRL2::INSTR.     |
| (GPIB VXI)?*INSTR      | Matches GPIB1::5::INSTR and VXI0::3::INSTR but not ASRL2::INSTR. |
| (GPIB0 VXI0)::1::INSTR | Matches GPIB0::1::INSTR and VXI0::1::INSTR.                      |
| ?*VXI[0-9]*::?*MEMACC  | Matches VXI0::MEMACC.  |
| VXI0::?*               | Matches VXI0::1::INSTR, VXI0::2::INSTR, and VXI0::MEMACC.        |
| ?*                     | Matches all resources.   |

Notice that in VISA, the regular expressions used for resource matching are not case sensitive. For example, calling `viFindRsrc()` with `"VXI?*INSTR"` would return the same resources as invoking it with `"vxi?*instr"`.

## Attribute-Based Resource Matching

VISA can also search for a resource based on the values of the resource's attributes. The `viFindRsrc()` search expression is handled in two parts: the regular expression for the resource string and the (optional) logical expression for the attributes. Assuming that a given resource matches the given regular expression, VISA checks the attribute expression for a match. The resource matches the overall string if it matches both parts.

Attribute matching works by using familiar constructs of logical operations such as AND (&&), OR (||), and NOT (!). Equal (==) and unequal (!=) apply to all types of attributes, and you can additionally compare numerical attributes using other common comparators (>, <, >=, and <=).

You are free to make attribute matching expressions as complex as you like, using multiple ANDs, ORs, and NOTs. Precedence applies as follows:

- The grouping operator ( ) in an attribute matching expression has the highest precedence.
- The NOT ! operator has the next highest precedence.

- The AND && operator has the next highest precedence.
- The OR operator || has the lowest precedence.

The following table shows three examples of matching based on attributes.

| Expression  | Meaning   |
|---|---|
| GPIB[0-9]*::?*::?*::INSTR<br>{VI_ATTR_GPIB_SECONDARY_ADDR > 0 &&<br>VI_ATTR_GPIB_SECONDARY_ADDR < 10} | Find all GPIB devices that have secondary addresses from 1 to 9.  |
| ASRL?*INSTR{VI_ATTR_ASRL_BAUD ==<br>9600}   | Find all serial ports configured at 9600 baud.  |
| ?*VXI?INSTR{VI_ATTR_MANF_ID ==<br>0xFF6 && !(VI_ATTR_VXI_LA ==0   <br>VI_ATTR_SLOT <= 0)}             | Find all VXI instrument resources with manufacturer ID of FF6 and which are not logical address 0, slot 0, or external controllers. |

Notice that only **global** VISA attributes are permitted in the attribute matching expression.

The following example uses a regular expression with attribute matching. Notice that because only the first match is needed, `VI_NULL` is passed for both the **retCount** and **findList** parameters. This tells VISA to automatically close the find list rather than return it to the application.

## C Example

```
#include <stdio.h>
#include "visa.h"

#define MANF_ID      0xFF6      /* 12-bit VXI manufacturer ID of device */
#define MODEL_CODE   0x0FE      /* 12-bit or 16-bit model code of device */

/* Find the first matching device and return a session to it */
ViStatus autoConnect2(ViSession instrSesn)
{
    ViStatus      status;
    ViSession      defaultRM, instr;
    ViChar        desc[VI_FIND_BUFLen], regExToUse[VI_FIND_BUFLen];
```

```

status = viOpenDefaultRM(&defaultRM);
if (status < VI_SUCCESS) {

    /* Error initializing VISA ... exiting */
    return status;
}
/* Find the first matching VXI instrument */
sprintf(regExToUse,
"?*VXI?*INSTR{VI_ATTR_MANF_ID==0x%x && VI_ATTR_MODEL_CODE==0x%x}", MANF_ID,
MODEL_CODE);
status = viFindRsrc(defaultRM, regExToUse, VI_NULL, VI_NULL, desc);
if (status < VI_SUCCESS) {

    /* Error finding resources ... exiting */
    viClose(defaultRM);
    return status;
}
status = viOpen(defaultRM, desc, VI_NULL, VI_NULL, &instr);
if (status < VI_SUCCESS) {

    viClose(defaultRM);
    return status;
}
*instrSesn = instr;
/* Do not close defaultRM, as that would close instr too */
return VI_SUCCESS;
}

```

## Visual Basic Example

This example uses functionality not available in Visual Basic.

### Related concepts:

- [Finding Resources](#)

## Configuring a Session

After the Resource Manager opens a session, communication with the device can usually begin using the default session settings. However, in some cases such as ASRL (serial) resources, you need to set some other parameters such as baud rate, parity, and flow control before proper communication can begin. GPIB and VXI sessions may

have still other configuration parameters to set, such as timeouts and end-of-transmission modes, although in general the default settings should suffice.

## Accessing Attributes

VISA uses two operations for obtaining and setting parameters—`viGetAttribute()` and `viSetAttribute()`. Attributes not only describe the state of the device, but also the method of communication with the device.

For example, you could use the following code to obtain the logical address of a VXI address:

```
status = viGetAttribute(instr, VI_ATTR_VXI_LA, &Laddr);
```

and the variable `Laddr` would contain the device's address. If you want to set an attribute, such as the baud rate of an ASRL session, you could use:

```
status = viSetAttribute(instr, VI_ATTR_ASRL_BAUD, 9600);
```

Notice that some attributes are read-only, such as logical address, while others are read/write attributes, such as the baud rate. Also, some attributes apply only to certain types of sessions; `VI_ATTR_VXI_LA` would not exist for an ASRL session. If you attempted to use it, the status parameter would return with the code `VI_ERROR_NSUP_ATTR`. Finally, the data types of some attribute values are different from each other. Using the above examples, the logical address is a 16-bit value, whereas the baud rate is a 32-bit value. It is particularly important to use variables of the correct data type in `viGetAttribute()`.

### Related concepts:

- [VISA Access Mechanisms](#)

## Common Considerations for Using Attributes

As you set up your sessions, there are some common attributes you can use that will affect how the sessions handle various situations. For currently supported session



types, all support the setting of timeout values and termination methods:

- `VI_ATTR_TMO_VALUE` denotes how long, in milliseconds, to wait for accesses to the device. Defaults to two seconds (2000 ms).
- `VI_ATTR_TERMCHAR_EN` sets whether a termination character specified by `VI_ATTR_TERMCHAR` will be used on read operations. The termchar defaults to linefeed (`\n` or `LF`) but the termchar enable attribute defaults to `VI_FALSE`. Serial users should also see `Serial`.
- `VI_ATTR_SEND_END_EN` determines whether to use an END bit on your write operations. Defaults to `VI_TRUE`.

Various interfaces have other types of attributes that may affect channel communication. See *Interface-Specific Information* for attribute information relevant to each support hardware interface type.

# VISA Message-Based Communication

Whether you are using RS-232, GPIB, Ethernet, VXI, or USB, message-based communication is a standard protocol for controlling and receiving data from instruments. Because most message-based devices have similar capabilities, it is natural that the driver interface should be consistent. Under VISA, controlling message-based devices is the same regardless of what hardware interface(s) those devices support or how those devices are connected to your computer.

VISA message-based communication includes the Basic I/O Services and the Formatted I/O Services from within the VISA Instrument Control Resource (INSTR). All sessions to a VISA Instrument Control Resource (INSTR) opened using `viOpen()` have full message-based communication capabilities. Of course, if the device is a register-based VXI device, the message-based operations return an error code (`VI_ERROR_NSUP_OPER`) to indicate that this device does not support the operations, although the **session** still provides access to them. This section discusses the uses of the Basic I/O Services and the Formatted I/O Services provided by the INSTR Resource in a VISA application, and shows how to use the VISA library in message-based communication.

## Related concepts:

- [Basic I/O Services](#)
- [Formatted I/O Services](#)

## Basic I/O Services

The VISA Instrument Control Resource lets a controller interact with the device that it is associated with by providing the controller with services to do the following:

- Send blocks of data to the device
- Request blocks of data from the device
- Send the device clear command to the device
- Trigger the device
- Find information about the status of the device



**Note** For the ASRL INSTR and TCPIP SOCKET resources, the I/O protocol attribute must be set to VI\_PROT\_4882\_STRS to use viReadSTB() and viAssertTrigger().

This section describes the operations provided by the VISA Instrument Control Resource for the Basic I/O Services.

## Synchronous Read/Write Services

The most straightforward of the operations are viRead() and viWrite(), which perform the actual receiving and sending of strings. Notice that these operations look upon the data as a string and do not interpret the contents. For this reason, the data could be messages, commands, or binary encoded data, depending on how the device has been programmed. For example, the IEEE 488.2 command \*IDN? is a message that is sent in ASCII format. However, an oscilloscope returning a digitized waveform may take each 16-bit data point and put it end to end as a series of 8-bit characters. The following code segment shows a program requesting the waveform that the device has captured.

```
status = viWrite(instr, "READ:WAVFM:CH1", 14, &retCount);  
status = viRead(instr, buffer, 1024, &retCount);
```

Now the character array `buffer` contains the data for the waveform, but you still do not know how the data is formatted. For example, if the data points were 1, 2, 3, ...the buffer might be formatted as "1,2,3,...". However, if the data were binary encoded 8-bit values, the first byte of `buffer` would be 1—not the ASCII character 1, but the actual value 1. The next byte would be neither a comma nor the ASCII character 2, but the actual value 2, and so on. Refer to the documentation that came with the device for information on how to program the device and interpret the responses.

The various ways that a string can be sent is the next issue to consider in message-based communication. For example, the actual mechanism for sending a byte differs drastically between GPIB and VXI; however, both have similar mechanisms to indicate when the last byte has been transferred. Under both systems, a device can specify an actual character, such as linefeed, to indicate that no more data will be sent. This is known as the End Of String (EOS) character and is common in older GPIB devices. The

obvious drawback to this mechanism is that you must send an extra character to terminate the communication, and you cannot use this character in your messages. However, both GPIB and VXI can specify that the current byte is the last byte. GPIB uses the EOI line on the bus, and VXI uses the END bit in the Word Serial command that encapsulates the byte.

You need to determine how to inform the VISA driver which mechanism to use. As was discussed in VISA Overview, VISA uses a technique known as attributes to hold this information. For example, to tell the driver to use the EOI line or END bit, you set the `VI_ATTR_SEND_END_EN` attribute to true.

```
status = viSetAttribute(instr, VI_ATTR_SEND_END_EN, VI_TRUE);
```

You can terminate reads on a carriage return by using the following code.

```
status = viSetAttribute(instr, VI_ATTR_TERMCHAR, 0x0D);  
status = viSetAttribute(instr, VI_ATTR_TERMCHAR_EN, VI_TRUE);
```

### Related concepts:

- [Common Considerations for Using Attributes](#)

## Asynchronous Read/Write Services

In addition to the synchronous read and write services, VISA has operations for asynchronous I/O. The functionality of these operations is identical to that of the synchronous ones; therefore, the topics covered in the previous section apply to asynchronous read and write operations as well. The main difference is that a job ID is returned from the asynchronous I/O operations instead of the transfer status and return count. You then wait for an I/O completion event, from which you can get that information.



**Note** You must enable the session for the I/O completion event before beginning an asynchronous transfer.

One other difference is the timeout attribute, `VI_ATTR_TMO_VALUE`. This attribute may

or may not apply to asynchronous operations, depending on the implementation. If you want to ensure that asynchronous operations never time out, even on implementations that do use the timeout attribute, set the attribute value to `VI_TMO_INFINITE`. If you want to ensure that asynchronous operations do not last beyond a certain period of time, even on implementations that do not use the timeout attribute, you should abort the I/O using the `viTerminate()` operation if it does not complete within the expected time, as shown in the following code.

```
status = viEnableEvent(instr, VI_EVENT_IO_COMPLETION, VI_QUEUE, VI_NULL);
status = viWriteAsync(instr, "READ:WAVFM:CH1" ,14, &jobID);
status = viWaitOnEvent(instr, VI_EVENT_IO_COMPLETION, 10000, &etype, &event);
if (status < VI_SUCCESS) {
    status = viTerminate(instr, VI_NULL, jobID);
    /* now the I/O completion event should exist in the queue*/
    status = viWaitOnEvent(instr, VI_EVENT_IO_COMPLETION,0, &etype, &event);
}
```

As long as an asynchronous operation is successfully posted (if the return value from the asynchronous operation is greater than or equal to `VI_SUCCESS`), there will always be exactly one I/O completion event resulting from the transfer. However, if the asynchronous operation (`viReadAsync()` or `viWriteAsync()`) returns an error code, there will not be an I/O completion event. In the above example, if the I/O has not completed in 10 seconds, the call to `viTerminate()` aborts the I/O and results in the I/O completion event being generated.

The I/O completion event has attributes containing information about the transfer status, return count, and more. For a more complete description of the I/O completion event and its attributes, refer to `VI_EVENT_IO_COMPLETION`. For a more detailed example using asynchronous I/O, see the queuing and callback mechanism example.



**Note** The asynchronous I/O services are not available when programming with Visual Basic.

### Related concepts:

- [Queuing and Callback Mechanism Sample Code](#)

## Clear Service

When communicating with a message-based device, particularly when you are first developing your program, you may need to tell the device to clear its I/O buffers so that you can start again. In addition, if a device has more information than you need, you may want to read until you have everything you need and then tell the device to throw the rest away. The `viClear()` operation performs these tasks.

More specifically, the clear operation lets a controller send the device clear command to the device it is associated with, as specified by the interface specification and the type of device. The action that the device takes depends on the interface to which it is connected.

- For a GPIB device, the controller sends the IEEE 488.1 SDC (04h) command.
- For a VXI or MXI device, the controller sends the Word Serial Clear (FFFFh) command.
- For the ASRL INSTR or TCPIP SOCKET resource, the controller sends the string `"*CLS\n"`. The I/O protocol must be set to `VI_PROT_4882_STRS` for this service to be available to these resources.

For more details on these clear commands, refer to your device documentation, the IEEE 488.1 standard, or the VXIbus specification.

## Trigger Service

Most instruments can be instructed to wait until they receive a trigger before they start performing operations such as generating a waveform, reading a voltage, and so on. Under GPIB, this trigger is a software command sent to the device. Under VXI, this could either be a software trigger or a hardware trigger on one of the multiple TTL/ECL trigger lines on the VXIbus backplane.

VISA uses the same operation—`viAssertTrigger()`—to perform these actions. Which trigger method (software or hardware) you use is dependent on a combination of an attribute (`VI_ATTR_TRIG_ID`) and a parameter to the operation. For example, to send a software trigger by default under either interface, you use the following code.

```
status = viSetAttribute(instr, VI_ATTR_TRIG_ID, VI_TRIG_SW);
status = viAssertTrigger(instr, VI_TRIG_PROT_DEFAULT);
```

Of course, you need to set the attribute only once at the beginning of the program, not every time you assert the trigger. If you want to assert a VXI hardware trigger, such as a SYNC pulse, you can use the following code.

```
status = viSetAttribute(instr, VI_ATTR_TRIG_ID, VI_TRIG_TTL3);
status = viAssertTrigger(instr, VI_TRIG_PROT_SYNC);
```

Keep in mind that VISA currently uses **device triggering**. That is, each call to `viAssertTrigger()` is associated with a specific device through the session used in the call. However, the VXI hardware triggers by definition have **interface-level triggering**. In other words, you cannot prevent two devices from receiving a SYNC pulse of TTL3 if both devices are listening to the line. Therefore, if you need to trigger multiple devices off a single VXI trigger line, you can do this by sending the trigger to any one of the devices on the line.

## Status/Service Request Service

It is fairly common for a device to need to communicate with a controller at a time when the controller is not planning to talk with the device. For example, if the device detects a failure or has completed a data acquisition sequence, it may need to get the attention of the controller. In both GPIB and VXI, this is accomplished through a Service Request (SRQ). Although the actual technique for delivering this service request to the controller differs between the two interfaces, the end result is that an event (`VI_EVENT_SERVICE_REQ`) is received by the VISA driver. You can find more details on event notification and handling in *Introductory Programming Examples* and *VISA Events*. At this time, just assume that the program has received the event and has a handle to the data through the `eventContext` parameter.

Under VISA, the `VI_EVENT_SERVICE_REQ` event contains no additional information other than the type of event. Therefore, by using `viGetAttribute()` on the `eventContext` parameter, as shown in the following code, the program can identify the event as a service request.

```
status = viGetAttribute(eventContext, VI_ATTR_EVENT_TYPE, &eventType);
```

You can retrieve the status byte of the device by issuing a `viReadSTB()` operation. This is especially important because on some interfaces, such as GPIB, it is not always possible to know which device has asserted the service request until a `viReadSTB()` is performed. This means that all sessions to devices on the bus with the service request may receive a service request event. Therefore, you should always check the status byte to ensure that your device was the one that requested service. Even if you have only one device asserting a service request, you should still call `viReadSTB()` to guarantee delivery of future service request events. For example, the following code checks the type of event, performs a `viReadSTB()`, and then checks the result.

```
status = viGetAttribute(eventContext, VI_ATTR_EVENT_TYPE, &eventType);
if (eventType == VI_EVENT_SERVICE_REQ) {
    status = viReadSTB(instr, &statusByte);
    if ((status >= VI_SUCCESS) && (statusByte & 0x40)) {
        /* Perform action based on Service Request */
    }
    /* Otherwise ignore the Service Request */
} /* End IF SRQ */
```

## Related concepts:

- [VISA Events](#)

## Example VISA Message-Based Application

The following is an example VISA application using message-based communication.

### C Example

```
#include "visa.h"
int main(void)
{
    ViSession    defaultRM, instr;
    ViUInt32     retCount;
    ViChar       idnResult[72];
    ViChar       resultBuffer[256];
    ViStatus     status;
```



```

/* Open Default Resource Manager */
status = viOpenDefaultRM(&defaultRM);
if (status < VI_SUCCESS) {
    /* Error Initializing VISA...exiting */
    return -1;
}
/* Open communication with GPIB Device at Primary Addr 1 */
/* NOTE: For simplicity, we will not show error checking */
viOpen(defaultRM, "GPIB::1::INSTR", VI_NULL, VI_NULL, &instr);
/* Initialize the timeout attribute to 10 s */
viSetAttribute(instr, VI_ATTR_TMO_VALUE, 10000);
/* Set termination character to carriage return (\r=0x0D) */
viSetAttribute(instr, VI_ATTR_TERMCHAR, 0x0D);
viSetAttribute(instr, VI_ATTR_TERMCHAR_EN, VI_TRUE);
/* Don't assert END on the last byte */
viSetAttribute(instr, VI_ATTR_SEND_END_EN, VI_FALSE);
/* Clear the device */
viClear(instr);
/* Request the IEEE 488.2 identification information */
viWrite(instr, "*IDN?\n", 6, &retCount);
viRead(instr, idnResult, 72, &retCount);

/* Use idnResult and retCount to parse device info */

/* Trigger the device for an instrument reading */
viAssertTrigger(instr, VI_TRIG_PROT_DEFAULT);
/* Receive results */
viRead(instr, resultBuffer, 256, &retCount);
/* Close sessions */
viClose(instr);
viClose(defaultRM);
return 0;
}

```

## Visual Basic Example

```

Private Sub vbMain()

    Dim stat          As ViStatus
    Dim dfltRM        As ViSession
    Dim sesn          As ViSession
    Dim retCount      As Long
    Dim idnResult     As String * 72
    Dim resultBuffer  As String * 256
    Rem Open Default Resource Manager

```

```

stat = viOpenDefaultRM(dfltRM)
If (stat < VI_SUCCESS) Then
    Rem Error initializing VISA...exiting
    Exit Sub
End If

Rem Open communication with GPIB Device at Primary Addr 1
Rem NOTE: For simplicity, we will not show error checking
stat = viOpen(dfltRM, "GPIB::1::INSTR", VI_NULL, VI_NULL, sesn)

Rem Initialize the timeout attribute to 10 s
stat = viSetAttribute(sesn, VI_ATTR_TMO_VALUE, 10000)

Rem Set termination character to carriage return (\r=0x0D)
stat = viSetAttribute(sesn, VI_ATTR_TERMCHAR, &H0D)
stat = viSetAttribute(sesn, VI_ATTR_TERMCHAR_EN, VI_TRUE)

Rem Don't assert END on the last byte
stat = viSetAttribute(sesn, VI_ATTR_SEND_END_EN, VI_FALSE)

Rem Clear the device
stat = viClear(sesn)

Rem Request the IEEE 488.2 identification information
stat = viWrite(sesn, "*IDN?", 5, retCount)
stat = viRead(sesn, idnResult, 72, retCount)

Rem Your code should use idnResult and retCount to parse device info

Rem Trigger the device for an instrument reading
stat = viAssertTrigger(sesn, VI_TRIG_PROT_DEFAULT)

Rem Receive results
stat = viRead(sesn, resultBuffer, 256, retCount)

Rem Close sessions
stat = viClose (sesn)
stat = viClose (dfltRM)

End Sub

```

## Formatted I/O Services

The Formatted I/O Services perform formatted and buffered I/O for devices. A formatted write operation writes to a buffer inside the VISA driver, while a formatted

read operation reads from a buffer inside the driver. Buffering improves system performance by having the driver perform the I/O with the device only at certain times, such as when the buffer is full. The driver is then able to send larger blocks of information to the device at a time, improving overall throughput.

The buffer operations also provide control over the low-level serial driver buffers.

### Related concepts:

- [Controlling the Serial I/O Buffers](#)

## Formatted I/O Operations

The main two operations under the formatted I/O services are `viPrintf()` and `viScanf()`. Although this section discusses these two operations only, this material also applies to other formatted I/O routines such as `viPrintf()` and `viScanf()`. These operations derive their names from the standard C string I/O functions. Like `printf()` and `scanf()`, these operations let you use special format strings to dynamically create or parse the string. For example, a common command for instruments is the "F $x$ " command for function  $x$ . This could be "F1" for volt measurement, "F2" for ohm measurement, and so on. With formatted I/O, you can select the type of measurement and use only a single operation to send the string. Consider the following code segment.

```
/* Retrieve user's selections. Assume the variable */
/* X holds the choice from the following menu: */
/* 1) VDC, (2) Ohms, (3) Amps */
status = viPrintf(instr, "F%d", X);
```

Here, the variable `x` corresponds to the type of measurement denoted by a number matching the function number for the instrument. Without formatted I/O, the result would have been either:

```
sprintf(buffer, "F%d", X);
viWrite(instr, buffer, strlen(buffer), &retCount);
```

or

```
switch(X) {
    case 1:
        viWrite(instr, "F1", 2, &retCount);
        break;
    case 2:
        viWrite(instr, "F2", 2, &retCount); break;
    .
    .
}
```

In addition, there is an operation `viQueryf()` that combines the functionality of a `viPrintf()` followed by a `viScanf()` operation. `viQueryf()` is used to query the device for information:

```
status = viQueryf(instr, "*IDN?\n", "%s", buf);
```

## I/O Buffer Operations

Another method for communicating with your instruments using formatted I/O functions is using the formatted I/O buffer functions: `viSPrintf()`, `viSScanf()`, `viBufRead()`, and `viBufWrite()`. You can use these functions to manipulate a buffer that you will send or receive from an instrument.

For example, you may want to bring information from a device into a buffer and then manipulate it yourself. To do this, first call `viBufRead()`, which reads the string from the instrument into a user-specified buffer. Then use `viSScanf()` to extract information from the buffer. Similarly, you can format a buffer with `viSPrintf()` and then use `viBufWrite()` to send it to an instrument.

As you can see, the formatted I/O approach is the simplest way to get the job done. Because of the variety of modifiers you can use in the format string, this topic does not go into any more detail on these operations.

## Variable List Operations

You can also use another form of the standard formatted I/O operations known as Variable List operations: `viVPrintf()`, `viVSPrintf()`, `viVScanf()`, `viVSScanf()`, and `viVQueryf()`. These functions are identical in their operation to the ANSI C versions of

variable list operations. See your C reference guide for more information.

## Manually Flushing the Formatted I/O Buffers

This section describes flushing issues that are related to formatted I/O buffers. The descriptions apply to all buffered read and buffered write operations. For example, the `viPrintf()` description applies equally to other buffered write operations (`viVPrintf()` and `viBufWrite()`). Similarly, the `viScanf()` description applies to other buffered read operations (`viVScanf()` and `viBufRead()`).

Flushing a write buffer immediately sends any queued data to the device. Flushing a read buffer discards the data in the read buffer. An empty read buffer guarantees that the next call to `viScanf()`, `viBufRead()`, or a related operation reads data directly from the device rather than from queued data residing in the read buffer.

The easiest way to flush the buffers is with an explicit call to `viFlush()`. This operation can actually flush the buffers in two ways. The simpler way uses discard flags. These flags tell the driver to discard the contents of the buffers without performing any I/O to the device. For example,

```
status = viFlush(instr, VI_READ_BUF_DISCARD);
```

However, the flush operation can also complete the current I/O before flushing the buffer. For a write buffer, this simply means to send the rest of the buffer to the device. However, for a read buffer, the process is more involved. Because you could be in the middle of a read from the device (that is, the device still has information to send), it is possible to have the driver check the buffer for an EOS or END bit/EOI signal. If such a value exists in the buffer, the contents of the buffer are discarded. However, if the driver can find no such value, it begins reading from the device until it detects the end of the communication and then discards the data. This process keeps the program and device in synchronization with each other.

### Related concepts:

- [Automatically Flushing the Formatted I/O Buffers](#)
- [Formatted I/O Read and Low-Level I/O Receive Buffers](#)
- [Formatted I/O Write and Low-Level I/O Transmit Buffers](#)

- [Recommendations for Using the VISA Buffers](#)

## Automatically Flushing the Formatted I/O Buffers

Although you can explicitly flush the buffers by making a call to `viFlush()`, the buffers are flushed implicitly under some conditions. These conditions vary for the `viPrintf()` and `viScanf()` operations. In addition, you can modify the conditions through attributes.

The write buffer is maintained by the `viPrintf()`, `viVPrintf()`, `viBufWrite()`, and `viVQueryf()` (write side) operations. To explicitly flush the write buffer, you can make a call to the `viFlush()` operation with a write flag set.

The standard conditions for automatically flushing the buffer are as follows.

- Whenever the END indicator is sent. The indicator could be either the EOS character or the END bit/EOI line, depending on the current state of the attributes which select these modes.
- When the write buffer is full.
- In response to a call to `viSetBuf()` with the `VI_WRITE_BUF` flag set.

In addition to these rules, the `VI_ATTR_WR_BUF_OPER_MODE` attribute can modify the flushing of the buffer. The default setting for this attribute is `VI_FLUSH_WHEN_FULL`, which means that the preceding three rules apply. However, if the attribute is set to `VI_FLUSH_ON_ACCESS`, the buffer is flushed with every call to `viPrintf()` and `viVPrintf()`, essentially disabling the buffering mode.

The read buffer is maintained by the `viScanf()`, `viVScanf()`, `viBufRead()`, and `viVQueryf()` (read side) operations. To explicitly flush the read buffer, you can make a call to the `viFlush()` operation with a read flag set. The only rule for automatically flushing the read buffer is in response to the `viSetBuf()` operation. However, as with the write buffer, you can use an attribute to control how to flush the buffer: `VI_ATTR_RD_BUF_OPER_MODE`. If the attribute is set to `VI_FLUSH_DISABLE`, the buffer is flushed only when an explicit call to `viFlush()` is made. If this attribute is set to `VI_FLUSH_ON_ACCESS`, the buffer is flushed at the end of every call to `viScanf()`.

In addition to the preceding rules and attributes, the formatted I/O buffers of a session to a given device are reset whenever that device is cleared through the `viClear()`

operation. At such a time, the read and write buffer must be flushed and any ongoing operation through the read/write port must be aborted.

### Related concepts:

- [Formatted I/O Read and Low-Level I/O Receive Buffers](#)
- [Formatted I/O Write and Low-Level I/O Transmit Buffers](#)
- [Manually Flushing the Formatted I/O Buffers](#)
- [Recommendations for Using the VISA Buffers](#)

## Resizing the Formatted I/O Buffers

### Resizing the Formatted I/O Buffers

The read and write buffers, as mentioned previously, can be dynamically resized using the `viSetBuf()` operation. Remember that this operation automatically flushes the buffers, so it is best to set the size of the buffers before beginning the actual I/O calls. You specify which buffer you want to modify and then the size of the buffer you require. It is important to check the return code of this operation because you may be requesting a buffer beyond the size that the system can allocate at the time. If this occurs, the buffer size is not changed.

For example, to set both the read and write buffers to 8 KB, use the following code.

```
status = viSetBuf(instr, VI_READ_BUF | VI_WRITE_BUF, 8192);
```

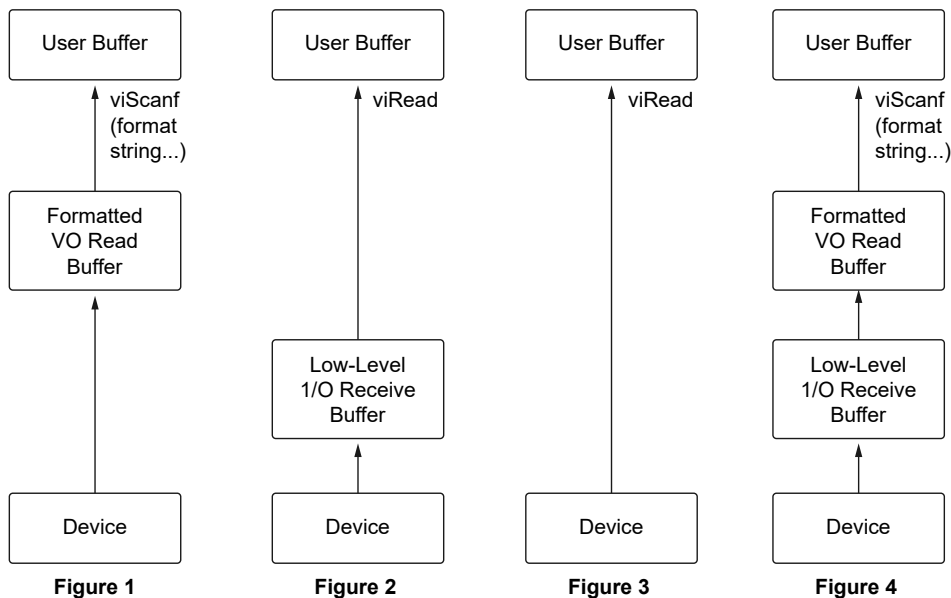
## Formatted I/O Read and Low-Level I/O Receive Buffers

VISA uses two basic input buffers: the formatted I/O read buffer and low-level I/O receive buffer. The formatted I/O read buffer eliminates the first-byte latency overhead that would exist if VISA performed its formatting algorithms by retrieving data from the device one byte at a time. Instead, VISA transfers a block of data from the device to the formatted I/O read buffer. VISA then performs its formatting algorithms on the raw data in the formatted I/O read buffer and places the formatted data into the user buffer, as shown in Figure 1. `viScanf` and its variants (for example, `viBufRead` and `viQueryf`) are the only operations that use the formatted I/O read buffer.

The low-level I/O receive buffer avoids data loss with resources that push or stream their response data to the host PC. Serial INSTR and TCPIP Socket are two examples of this type of resource. `viRead` and its variants (for example, `viReadAsync`) transfer data from the low-level I/O receive buffer into the user buffer in this scenario, as shown in Figure 2.

In contrast, when VISA must request data from a resource (such as GPIB INSTR), a call to `viRead` (or one of its variants) transfers data from the device directly into the user buffer, as shown in Figure 3.

The only time when both the formatted I/O read buffer and low-level I/O receive buffer are used is when `viScanf` is called on a resource that pushes data to the host PC (such as Serial INSTR or TCPIP Socket), as shown in Figure 4.



### Related concepts:

- [Automatically Flushing the Formatted I/O Buffers](#)
- [Controlling the Serial I/O Buffers](#)
- [Formatted I/O Write and Low-Level I/O Transmit Buffers](#)
- [Manually Flushing the Formatted I/O Buffers](#)
- [Recommendations for Using the VISA Buffers](#)

## Formatted I/O Write and Low-Level I/O Transmit Buffers

VISA uses two basic output buffers: the formatted I/O write buffer and low-level I/O



transmit buffer. The formatted I/O write buffer holds the converted and formatted parameters (as the format string specifies) before sending the formatted data to the device, as shown in Figure 1. `viPrintf` and its variants (for example, `viBufWrite` and `viQueryf`) are the only operations that use the formatted I/O write buffer.

The advantage of using formatted I/O buffers for data to be written to a device is that if you are sending multiple small formatted strings in several programmatic commands, you can batch them and have VISA send them in a single transfer; this eliminates repeating the first-byte latency overhead. On the other hand, if you do not need VISA to format or buffer your data, you can send it directly to the device from the user's data buffer using `viWrite` and its variants (for example, `viWriteAsync`), as shown in Figure 2.

The VISA specification defines the low-level I/O transmit buffer, but it is rarely used in practice. Its main purpose is to provide logical parallelism. Therefore, except in the case of serial resources, the low-level I/O transmit buffer is not needed, and data is sent directly to the device, as shown in Figure 2.

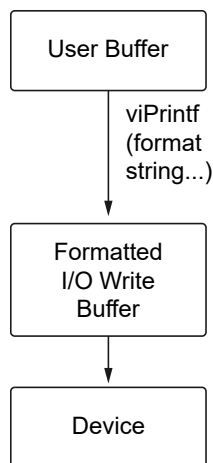


Figure 1

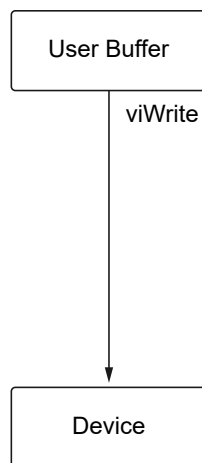


Figure 2

### Related concepts:

- [Controlling the Serial I/O Buffers](#)
- [Automatically Flushing the Formatted I/O Buffers](#)
- [Formatted I/O Read and Low-Level I/O Receive Buffers](#)
- [Manually Flushing the Formatted I/O Buffers](#)
- [Recommendations for Using the VISA Buffers](#)

## Recommendations for Using the VISA Buffers

Unless you are dealing with large buffers, it is usually unnecessary to call `viSetBuf` to adjust the size of the VISA buffers.

Because the low-level I/O transmit buffer is rarely used, there is seldom, if ever, a need to call `viSetBuf` with the `VI_IO_OUT_BUF` flag. Even in the case of serial resources, changing the low-level I/O transmit buffer size may have no effect.

In general, calling `viFlush` with the `VI_IO_OUT_BUF` or `VI_IO_OUT_BUF_DISCARD` flags has no effect.

If you need to call `viFlush` on the low-level I/O receive buffer, the data is always discarded, regardless of whether you specify `VI_IO_IN_BUF` or `VI_IO_IN_BUF_DISCARD`.

When using formatted I/O, use `viClear` to discard the formatted I/O buffers, when possible, instead of `viFlush`. This ensures that both the read and write formatted I/O buffers, as well as the device's internal buffers, are cleared. Calling just `viFlush` to discard the formatted I/O buffers may leave unread data on the device, which leaves the device in an unknown state. In other words, further calls to `viPrintf` or `viScanf` may return unpredictable results.

In cases when you need to push the remaining formatted I/O write buffer contents to the device, call `viFlush` with the `VI_WRITE_BUF` flag. The formatted I/O write buffer is automatically flushed under certain conditions.

### Related concepts:

- [Automatically Flushing the Formatted I/O Buffers](#)
- [Formatted I/O Read and Low-Level I/O Receive Buffers](#)
- [Formatted I/O Write and Low-Level I/O Transmit Buffers](#)
- [Manually Flushing the Formatted I/O Buffers](#)

## Formatted I/O Instrument Driver Examples

This section shows examples of VISA formatted I/O usage found in existing instrument

drivers. These examples show how to perform various I/O tasks using the formatted I/O services in VISA. They assume a basic knowledge of string formatting and ANSI-C format specifiers. For more information on VISA format specifiers, refer to the appropriate topics in this help file.

The VISA formatting capabilities include those specified by ANSI-C with extensions for common protocols used by instrumentation systems. To perform I/O, use the `viPrintf()`, `viScanf()`, and `viQueryf()` service routines with the appropriate format strings.

For each category of formatted I/O, we give a description and a list of short examples. The focus is on the VISA I/O supported format specifiers that are most frequently used in driver development, with an explanation of how different modifiers work with the format codes. To eliminate redundancy and make the examples easier to understand, we have omitted error-checking routines on I/O operations from all examples.

## Integers

Integer formatting is often found in driver development. Besides transferring the numeric values that the instrument reads, it may also represent the status codes (Boolean values) or error codes returned by the instrument. When writing integer values to or reading them from the instrument, you can use `%d` format code with length modifiers (`h`, `l`, and `ll`) and the array modifier (`,`).

### Short Integer—"`%hd`"

Use this modifier for short (16 bit) integers. These are typically used for holding test results and status codes.

### Examples of Short Integer—"`%hd`"

This example shows how to scan a self test result (a 16 bit integer) returned from an instrument into a short integer.

```
/* Self Test */
ViInt16 testResult;
viPrintf (io, "*TST?\n");
viScanf (io, "%hd", testResult);
/* read test result into short integer */
```

This example shows how to query the instrument to determine whether it has encountered an error. The error status is returned as a short integer (16 bits).

```
/* Check Error Status */
ViInt16 esr;
viQueryf (io, "*ESR?\n", "%hd", &esr);
/* read status into short integer */
```

## Long Integer—" %ld", "%d"

Use this modifier for long (32 bit) integers. These are typically used for data value transfers and error code queries.

## Examples of Long Integer—" %ld", "%d"

This example shows how to scan an error code (a 32 bit integer) returned from an instrument into a 32 bit integer.

```
/* Error query */
ViInt32 errCode;
viPrintf (io, ":STAT:ERR?\n");
viScanf (io, "%d", &errCode);
/* read error code into integer */
```

This example shows how to format the sample count (a 32 bit integer) into the command string sent to an instrument.

```
/* Send Sample Count */
ViInt32 value = 5000;
viPrintf (io, ":SAMP:COUN %d;", value);
```

## Floating Point Values

When writing floating point values to or reading them from the instrument, you can use %f or %e format codes with length modifiers (l and L) and the array modifier (, ). Floating point values are important when programming a numeric value transfer.



**Note** `%f` does not fully represent a floating point value in the extreme cases. Use `%e` for a floating point value in such cases.

## Double Float—"`%le`"

Use this modifier for double (64 bit) floats. These are typically used for data value transfers.

### Examples of Double Float—"`%le`"

This example shows how to scan the vertical range (a 64 bit floating point number).

```
/* Query Vertical Range */
ViReal64 value;
viPrintf (io, ":CH1:SCA?\n");
viScanf (io, "%le", &value);
```

This example shows how to format a trigger delay of 50.0 (specified as a 64 bit floating point number) into the command string sent to an instrument.

```
/* Send Trigger Delay */
ViReal64 value = 50.0;
viPrintf (io, ":TRIG:DEL %le;", value);
```

## Precision Specifier—"`."`"

Use the precision specifier to specify the number of precision digits when doing a numeric transfer. This modifier sets the accuracy of the values.

### Examples of Precision Specifier—"`."`"

This example shows how to set the voltage resolution. The resolution is represented in a double floating point (64 bits). The precision modifier `.9` specifies that there are nine digits after the decimal point. In this case, 0.000000005 is sent to the instrument.

```
/* Set Resolution */
ViReal64 value = 0.0000000051;
viPrintf (io, "VOLT:RES %.9le", value);
```

## Array of Floating Point Values Specifier—"."

Use this modifier when transferring an array of floating point values to or from an instrument. The count of the number of elements can be represented by a constant, asterisk (\*) sign, or number (#) sign. The asterisk (\*) sign indicates the count is the first argument on `viPrintf()`. The number (#) sign indicates that the count is the first argument on `viScanf()`, and the count is passed by address. You can use the constant with both `viPrintf()` and `viScanf()`.

## Examples of Array of Floating Point Values Specifier—"."

This example shows how to send an array of double numbers to the instrument. The comma (,) indicates the parameter is an array and the asterisk (\*) specifies the array size to be passed in from the argument.

```
/* Create User Defined Mask */
ViInt32 maskSize = 100;
ViReal64 interleaved[100];
/* define points in the specified mask and store them in the array */
viPrintf (io, ":MASK:MASK1:POINTS %*,le", maskSize, interleaved);
```

This example shows how to take multiple readings from an instrument. The comma (,) indicates the parameter is an array and the number (#) sign specifies the actual number of readings returns from the instrument.

```
/* Read Multi-Point */
ViInt32 readingCnt = 50;
ViReal64 readingArray[50];
viQueryf (io, "READ?\n", "%,#le", &readingCnt, readingArray);
```

This example shows how to fetch multiple readings from an instrument. The comma (,) indicates the parameter is an array while the constant 1000 specifies the number of readings.

```
/* Fetch Multi-Point */
ViReal64 readingArray[1000];
viScanf (io, "%,1000le", readingArray);
```

## Strings

When transferring string values to or from the instrument, you can use `%s`, `%t`, `%T`, and `%[ ]` format codes with a field width modifier. Because this is a message-based communication system, string formatting is the most common routine. With string formatting, you can configure instrument settings and query instrument information.

### White Space Termination—"`%s`"

Characters are read from an instrument into the string until a white space character is read.

#### Examples of White Space Termination—"`%s`"

This example queries the trigger source. This instrument returns a string. The maximum length of the string is specified in the format string with the number (`#`) sign. The argument `rdBufferSize` contains the maximum length on input, and it contains the actual number of bytes read on output.

```
/* Trigger Source Query */
ViChar rdBuffer[BUFFER_SIZE];
ViInt32 rdBufferSize = sizeof(rdBuffer);
viPrintf (io, ":TRIG:SOUR?\n");
viScanf (io, "%#s", &rdBufferSize, rdBuffer);
```

### END Termination—"`%t`"

Characters are read from an instrument into the string until the first END indicator is received. This will often be accompanied by the linefeed character (`\n`), but that is not always the case. Use `%T` to parse up to a linefeed instead of an END.

#### Examples of END Termination—"`%t`"

This example queries the instrument model on a Tektronix instrument. The model number, a 32-bit integer, is the part of the string between the first two characters `", "` returned from the instrument. The format string `%t` specifies that the string reads from the device until the END indicator is received. For instance, if the instrument returns `TEKTRONIX, TDS 210, 0, CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04\n`,

the model number is 210, and the module string is 0,CF:91.1CT FV:v1.16  
TDS2CM:CMV:v1.04\n.

```
/* Instrument Model Information */
ViChar moduleStr[BUFFER_SIZE];
ViInt32 modelNumber;
viPrintf (io, "*IDN?\n");
viScanf (io, "TEKTRONIX,TDS %d,%t", &modelNumber, moduleStr);
```

## Other Terminators—" %[^]", "%\*[^]"

Without the asterisk, characters are read from an instrument into the string until the character specified after ^ is read. With the asterisk, characters are discarded until the character specified after ^ is read.

## Examples of Other Terminators—" %[^]", "%\*[^]"

This is an example of how to perform a self-test. In this case, the format string %256[^\n] specifies the maximum field width of the string as 256 and terminates with a line feed (LF) character.

```
/* Self Test */
ViChar testMessage[256];
viPrintf (io, "TST\n");
viScanf (io, "%256[^\n]", testMessage);
```

This example shows how to query for an error. The instrument returns an integer (32 bits) as the error code and a message that terminates with a double-quote ("). The message is in quotes.

```
/* Error Query */
ViInt32 errCode;
ViChar errMessage[MAX_SIZE];
viPrintf (io, ":STAT:ERR?\n");
viScanf (io, "%d,\"%[^\"]\"", &errCode, errMessage);
```

This example shows how to query for the instrument manufacturer. The manufacturer name is the first part of the string, up to the character ", ", returned from the instrument. For instance, if the instrument returns



ROHDE&SCHWARZ,NRVD,835430/066,V1.52 V1.40\n, the manufacturer name is ROHDE&SCHWARZ. The rest of the response is discarded.

```
/* Instrument Manufacturer */
ViChar rdBuffer[256];
viQueryf (io, "*IDN?\n", "%256[^\,]*T", rdBuffer);
```

This example shows how to query for the instrument model. The model name is the part of the string between the first two characters ", " returned from the instrument. For instance, if the instrument returns ROHDE&SCHWARZ,NRVD, 835430/066,V1.52 V1.40\n, the model name is NRVD. The format string %\* [^, ] discards the input up to character ", ". The final part of the response is also discarded.

```
/* Instrument Model Information */
ViChar rdBuffer[256];
viQueryf (io, "*IDN?\n", "%*[^,],%256[^\,]*T", rdBuffer);
```

This example queries the instrument firmware revision. The firmware revision information is everything up to the carriage return (CR) character.

```
/* Instrument Firmware Revision */
ViChar rdBuffer[256];
viQueryf (io, "ROM?", "%256[^\r]", rdBuffer);
```

## Data Blocks

Both raw data and binary data can be transferred between the driver and the instrument. Data block transfer is a simple yet powerful formatting technique for transferring waveform data between drivers and instruments.

### IEEE 488.2 Binary Data—" %b"

When writing binary data to or reading it from the instrument, you can use %b, %B format codes with length modifiers (h, l, ll, z, and Z). ASCII data is represented by signed integer values. The range of values depends on the byte width specified. One-byte-wide data ranges from -128 to +127. Two-byte-wide data ranges from -32768 to +32767. An example of an ASCII waveform data string follows:

```
CURVE -109, -110, -109, -107, -109, -107, -105, -103, -100, -97, -90, -84, -80
```

## Examples of IEEE 488.2 Binary Data—"%"b"

This example queries a waveform. The data is in IEEE 488.2 <ARBITRARY BLOCK PROGRAM DATA> format. The number (#) sign specifies the data size. In the absence of length modifiers, the data is assumed to be of byte-size elements.

```
/* Waveform Query */
ViInt32 totalPoints = MAX_DATA_PTS;
ViInt8 rdBuffer[MAX_DATA_PTS];
viQueryf (io, ":CURV?\n", "%#b", &totalPoints, rdBuffer);
```

This example shows how to scan the preamble of waveform data returned from a scope, how to determine the number of data points in the waveform, and how to scan the array of raw binary data returned.

```
/* Waveform Preamble */
ViByte data[MAX_WAVEFORM_SIZE];
ViInt32 i, tmpCount, acqType;
ViReal64 xInc, xOrg, xRef, yInc, yOrg, yRef;

viQueryf (io, "WAV:PRE?\n",

"%^[^,], %ld, %ld, %^[^,], %Lf, %Lf, %Lf, %Lf, %Lf, %Lf",
&acqType, &tmpCount, &xInc, &xOrg, &xRef, &yInc, &yOrg, &yRef);

tmpCount = (acqType == 3) ? 2*tmpCount : tmpCount;
viQueryf (io, "WAV:DAT?\n", "%#b", &tmpCount, data));
```

## Raw Binary Data—"%"y"

When transferring raw binary data to or from an instrument, use the %y format code with length modifiers (h, l, and ll) and byte ordering modifiers (!ob and !ol). Raw binary data can be represented by signed integer values or positive integer values. The range of the values depends on the specified byte width:

| Byte Width | Signed Integer Range | Positive Integer Range |
|------------|----------------------|------------------------|
| 1          | -128 to +127         | 0 to 255               |

| Byte Width | Signed Integer Range | Positive Integer Range |
|------------|----------------------|------------------------|
| 2          | -32768 to +32767     | 0 to 65535             |

## Examples of Raw Binary Data—"%"

This example shows how to send a block of unsigned short integer (16 bits) in binary form to the instrument. In this case, the binary data is an arbitrary waveform. The asterisk (\*) specifies the block size to be passed in from the argument. Also, !ob specifies data is sent in standard (big endian) format. Use !ol to send data in little endian format.

```
/* Create Arbitrary Waveform */
ViInt32 wfmSize = WFM_SIZE;
ViUInt16 dataBuffer[WFM_SIZE]; /* contains waveform data */
dataBuffer[WFM_SIZE-1] |= 0x1000;
/* Add the end of waveform indicator */
viPrintf (io, "STARTBIN 0 %d;%*!obhy", wfmSize, wfmSize, dataBuffer);
```

This example shows how to send a block of signed integers (32 bits) in binary form to the instrument. The asterisk (\*) specifies the block size to be passed in from the argument. Without the presence of a byte order modifier, data is sent in standard (big endian) format.

```
/* Create FM Modulation Waveform */
ViInt32 dataBuffer[WFM_SIZE];
/*contains waveform data */
viPrintf (io, "%*ly", wfmSize, dataBuffer);
```

# VISA Register-Based Communication



**Note** Register-based programming applies only to PXI and VXI.

Register-based devices (RBDs) are a class of devices that are simple and relatively inexpensive to manufacture. Communication with such devices is usually accomplished via reads and writes to registers. VISA has the ability to read from and write to individual device registers, as well as a block of registers, through the Memory I/O Services.

In addition to accessing RBDs, VISA also provides support for memory management of the memory exported by a device. For example, both local controllers and remote devices can have general-purpose memory in A24/A32 space. With VISA, although the user must know how each remote device accesses its own memory, the memory management aspects of local controllers are handled through the Shared Memory operations—`viMemAlloc()` and `viMemFree()`.

With the Memory I/O Services, you access the device registers based on the session to the device. In other words, if a session communicates with a device at VXI logical address 16, you cannot use Memory I/O Services on that session to access registers on a device at any other logical address. The range of address locations you can access with Memory I/O Services on a session is the range of address locations assigned to that device. This is true for both High-Level and Low-Level Access operations.

To facilitate access to the device registers for multiple VXI devices, VISA allows you to open a VXI MEMACC (memory access) session. A session to a VXI MEMACC Resource allows an application to access the entire VXI memory range for a specified address space. The MEMACC Resource supports the same high-level and low-level operations as the INSTR Resource. Programmatically, the main difference between a VXI INSTR session and a VXI MEMACC session is the value of the offset parameter you pass to the register based operations. When using an INSTR Resource, all address parameters are relative to the device's assigned memory base in the given address space; knowing a device's base address is neither required by nor relevant to the user. When using a MEMACC Resource, all address parameters are absolute within the given address space; knowing a device's base address is both required by and relevant to the user.



**Note** A session to a MEMACC Resource supports only the high-level, low-level, and resource template operations. A MEMACC session does not support the other INSTR operations.

In VISA, you can choose between two styles for accessing registers—High-Level Access or Low-Level Access. Both styles have operations to read the value of a device register and write to a device register, as shown in the following table. In addition, there are high-level operations designed to read or write a block of data. The block-move operations do not have a low-level counterpart.

|       | High-Level Access   | High-Level Block  | Low-Level Access   |
|-------|---|---|--|
| Read  | <ul style="list-style-type: none"> <li>• <code>viIn8()</code></li> <li>• <code>viIn16()</code></li> <li>•</li> </ul>                          | <ul style="list-style-type: none"> <li>• <code>viMoveIn8()</code></li> <li>• <code>viMoveIn16()</code></li> <li>• <code>viMoveIn32()</code></li> </ul>    | <ul style="list-style-type: none"> <li>• <code>viPeek8()</code></li> <li>• <code>viPeek16()</code></li> <li>• <code>viPeek32()</code></li> </ul> |
| Write | <ul style="list-style-type: none"> <li>• <code>viOut8()</code></li> <li>• <code>viOut16()</code></li> <li>• <code>viOut32()</code></li> </ul> | <ul style="list-style-type: none"> <li>• <code>viMoveOut8()</code></li> <li>• <code>viMoveOut16()</code></li> <li>• <code>viMoveOut32()</code></li> </ul> | <ul style="list-style-type: none"> <li>• <code>viPoke8()</code></li> <li>• <code>viPoke16()</code></li> <li>• <code>viPoke32()</code></li> </ul> |



**Note** The other register-based communication topics use XX in the names of some operations to denote that the information applies to 8-bit, 16-bit, and 32-bit reads and writes. For example, `viInXX()` refers to `viIn8()`, `viIn16()`, and `viIn32()`.

## High-Level Access Operations

The High-Level Access (HLA) operations `viInXX()` and `viOutXX()` have a simple and easy-to-use interface for performing register-based communication. The HLA operations in VISA are wholly self-contained, in that all the information necessary to carry out the operation is contained in the parameters of the operation. The HLA operations also perform all the necessary hardware setup as well as the error detection and handling. There is no need to call other operations to do any other activity related to the register

access. For this reason, you should use HLA operations if you are just becoming familiar with the system.

To use `viInXX()` or `viOutXX()` operations to access a register on a device, you need to have the following information about the register:

- The address space where the register is located. In a VXI interface bus, for example, the address space can be A16, A24, or A32. In the PXI bus, the device's address space can be the PXI configuration registers or one of the BAR spaces (BAR0-BAR5).
- The offset of the register relative to the device for the specified address space. You do not need to know the actual base address of the device, just the offset.



**Note** When using the VXI MEMACC Resource, you need to provide the absolute VXI address (base + offset) for the register.

The following sample code reads the Device Type register of a VXI device located at offset 0 from the base address in A16 space, and writes a value to the A24 shared memory space at offset 0x20 (this offset has no special significance).

```
status = viIn16(instr, VI_A16_SPACE, 0, &retValue);
status = viOut16(instr, VI_A24_SPACE, 0x20, 0x1234);
```

With this information, the HLA operations perform the necessary hardware setup, perform the actual register I/O, check for error conditions, and restore the hardware state. To learn how to perform these steps individually, see the Low-Level Access operations.

The HLA operations can detect and handle a wide range of possible errors. HLA operations perform boundary checks and return an error code (`VI_ERROR_INV_OFFSET`) to disallow accesses outside the valid range of addresses that the device supports. The HLA operations also trap and handle any bus errors appropriately and then report the bus error as `VI_ERROR_BERR`.

That is all that is really necessary to perform register I/O. For more examples of HLA register I/O, see the register-based communication example.

**Related concepts:**

- [Example of Register-Based Communication](#)

## High-Level Block Operations

The high-level block operations `viMoveInXX()` and `viMoveOutXX()` have a simple and easy-to-use interface for reading and writing blocks of data residing at either the same or consecutive (incrementing) register addresses. Like the high-level access operations, the high-level block operations can detect and handle many errors and do not require calls to the low-level mapping operations. Unlike the high-level access operations, the high-level block operations do not have a direct low-level counterpart. To perform block operations using the low-level access operations, you must map the desired region of memory and then perform multiple `viPeekXX()` or `viPokeXX()` operation invocations, instead of a single call to `viMoveInXX()` or `viMoveOutXX()`.

To use the block operations to access a device, you need to have the following information about the registers:

- The address space where the registers are located. In a VXI interface, for example, the address space can be A16, A24, or A32. In the PXI bus, the device's address space can be the PXI configuration registers or one of the BAR spaces (BAR0-BAR5).
- The beginning offset of the registers relative to the device for the specified address space.



**Note** With an INSTR Resource, you do not need to know the actual base address of the device, just the offset.

- The number of registers or register values to access.

The default behavior of the block operations is to access consecutive register addresses. However, you can change this behavior using the attributes `VI_ATTR_SRC_INCREMENT` (for `viMoveInXX()`) and `VI_ATTR_DEST_INCREMENT` (for `viMoveOutXX()`). If the value is changed from 1 (the default value, indicating consecutive addresses) to 0 (indicating that registers are to be treated as FIFOs), then the block operations perform the specified number of accesses to the same register address.



**Note** The range value of 0 for the VI\_ATTR\_SRC\_INCREMENT and VI\_ATTR\_DEST\_INCREMENT attributes may not be supported on all VISA implementations. In this case, you may need to perform a manual FIFO block move using individual calls to the high-level or low-level access operations.

If you are using the block operations in the default mode (consecutive addresses), the number of elements that you want to access may not go beyond the end of the device's memory in the specified address space.

In other words, the following code sample reads the VXI device's entire register set in A16 space:

```
status = viMoveIn16(instr, VI_A16_SPACE, 0, 0x20, regBuffer16);
```

Notice that although the device has 0x40 bytes of registers in A16 space, the fourth parameter is 0x20. Why is this? Since the operation accesses 16-bit registers, the actual range of registers read is 0x20 accesses times 2 B, or all 0x40 bytes. Similarly, the following code sample reads a PXI device's entire register set in configuration space:

```
status = viMoveIn32 (instr, VI_PXI_CFG_SPACE, 0, 64, regBuffer32);
```

When using the block operations to access FIFO registers, the number of elements to read or write is not restricted, because all accesses are to the same register and never go beyond the end of the device's memory region. The following sample code writes 4 KB of data to a device's FIFO register in A16 space at offset 0x10 (this offset has no special significance):

```
status = viSetAttribute(instr, VI_ATTR_DEST_INCREMENT, 0);
status = viMoveOut32(instr, VI_A16_SPACE, 0x10, 1024, regBuffer32);
```

## Low-Level Access Operations

Low-Level Access (LLA) operations provide a very efficient way to perform register-based communication. LLA operations incur much less overhead than HLA operations for certain types of accesses. LLA operations perform the same steps that the HLA



operations do, except that each individual task performed by an HLA operation is an individual operation under LLA.

## Bus Errors

The LLA operations do not report bus errors. In fact, `viPeekXX()` and `viPokeXX()` do not report any error conditions. When using the LLA operations, you must ensure that the addresses you are accessing are valid.

## Overview of Register Accesses from Computers

Before learning about the LLA operations, first consider how a computer can perform a register access to an external device. There are two possible ways to perform this access. The first and more obvious, although primitive, is to have some hardware on the computer that communicates with the external device.

You would have to follow these steps:

1. Write the address you want.
2. Specify the data to send.
3. Send the command to perform the access.

As you can see, this method involves a great deal of communication with the local hardware.

The NI MXI plug-in cards and embedded VXI computers use a second, much more efficient method. This method involves taking a section of the computer's address space and **mapping** this space to another space, such as the VXI A16 space. Most PXI devices also have registers that are memory mapped into your computer.

To understand how mapping works, you must first remember that memory and address space are two different things. For example, most 32-bit CPUs have 4 GB of **address space**, but have **memory** measured in megabytes. This means that the CPU can put out over  $2^{32}$  possible addresses onto the local bus, but only a small portion of that corresponds to memory. In most cases, the memory chips in the computer will respond to these addresses. However, because there is less memory in the computer than address space, National Instruments can add hardware that

responds to other addresses. This hardware can then modify the address, according to the **mapping** that it has, to a VXI address and perform the access on the VXIbus automatically. The result is that the computer acts as if it is performing a local access, but in reality the access has been mapped out of the computer and to the VXIbus.

You may wonder what the difference is between the efficient method and the primitive method. They seem to be telling the hardware the same information. However, there are two important differences. In the primitive method, the communication described must take place for each access. However, the efficient method requires only occasional communication with the hardware. Only when you want a different address space or an address outside of the window do you need to reprogram the hardware. In addition, when you have set up your hardware, you can use standard memory access methods, such as pointer dereferences in C, to access the registers.

## Using VISA to Perform Low-Level Register Accesses

The first LLA operation you need to call to access a device register is the `viMapAddress()` operation, which sets up the hardware window and obtains the appropriate pointer to access the specified address space. The `viMapAddress()` operation first programs the hardware to map local CPU addresses to hardware addresses as described in the previous section. In addition, it returns a pointer that you can use to access the registers.

The following code is an example of programming the VXI hardware to access A16 space.

```
status = viMapAddress(instr, VI_A16_SPACE, 0, 0x40, VI_FALSE, VI_NULL, &address);
```

This sample code sets up the hardware to map A16 space, starting at offset 0 for 0x40 bytes, and returns the pointer to the window in `address`. Remember that the offset is relative to the base address of the device we are talking to through the `instr` session, not from the base of A16 space itself. Therefore, offset 0 does not mean address 0 in A16 space, but rather the starting point of the device's A16 memory. You can ignore the `VI_FALSE` and `VI_NULL` parameters for the most part because they are reserved for definition by a future version of VISA.

If you call `viMap Address()` on an INSTR session with an address space the device does

not support, or an offset or size greater than the device's memory range, then the VISA driver will not map the memory and will return an error.



**Note** To access the device registers through a VXI MEMACC session, you need to provide the absolute VXIbus addresses (base address for device + register offset in device address space).

If you need more than a single map for a device, you must open a second session to the device, because VISA currently supports only a single map per session. There is very low overhead in having two sessions because sessions themselves do not take much memory. However, you need to keep track of two session handles. Notice that this is different from the maximum number of windows you can have on a system. The hardware for the controller you are using may have a limit on the number of unique windows it can support.

When you are finished with the window or need to change the mapping to another address or address space, you must first unmap the window using the `viUnmapAddress()` operation. All you need to specify is which session you used to perform the map.

```
status = viUnmapAddress(instr);
```

## Operations Versus Pointer Dereference

After the `viMapAddress()` operation returns the pointer, you can use it to read or write registers. VISA provides the `viPeekXX()` and `viPokeXX()` operations to perform the accesses. On many systems, the `viMapAddress()` operation returns a pointer that you can also dereference directly, rather than calling the LLA operations. The performance gain achievable by using pointer dereferences over operation invocations is extremely system dependent. To determine whether you can use a pointer dereference to perform register accesses on a given mapped session, examine the value of the `VI_ATTR_WIN_ACCESS` attribute. If the value is `VI_DEREF_ADDR`, it is safe to perform a pointer dereference.

To make your code portable across different platforms, we recommend that you always use the accessor operations—`viPeekXX()` and `viPokeXX()`—as a backup method

to perform register I/O. In this way, not only is your source code portable, but your executable can also have binary compatibility across different hardware platforms, even on systems that do not support direct pointer dereferences:

```
viGetAttribute(instr, VI_ATTR_WIN_ACCESS, &access);
if (access == VI_DEREF_ADDR)
    *address = 0x1234;
else
    viPoke16(instr, address, 0x1234);
```

## Manipulating the Pointer

Every time you call `viMapAddress()`, the pointer you get back is valid for accessing a region of addresses. Therefore, if you call `viMapAddress()` with `mapBase` set to address 0 and `mapSize` to 0x40 (the configuration register space for a VXI device), you can access not only the register located at address 0, but also registers in the same vicinity by manipulating the pointer returned by `viMapAddress()`. For example, if you want to access another register at address 0x2, you can add 2 to the pointer. You can add up to and including 0x3F to the pointer to access these registers in this example because we have specified 0x40 as the map size. However, notice that you cannot subtract any value from the `address` variable because the mapping starts at that location and cannot go backwards. The following example shows how you can access other registers from `address`.

### C Example

```
#include "visa.h"
#define ADD_OFFSET(addr, offs) (((ViPByte)addr) + (offs))
int main(void)
{
    ViStatus    status;           /* For checking errors */
    ViAddr      defaultRM, instr; /* Communication channels */
    ViUInt16    address;         /* User pointer */
    ViSession    value;          /* To store register value */

    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error Initializing VISA...exiting */
        return -1;
    }
}
```

```

}

/* Open communication with VXI Device at Logical Address 16 */
/* NOTE: For simplicity, we will not show error checking */
status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &instr);

status = viMapAddress(instr, VI_A16_SPACE, 0, 0x40, VI_FALSE, VI_NULL,
&address);

viPeek16(instr, address, &value);
/* Access a different register by manipulating the pointer. */
viPeek16(instr, ADD_OFFSET(address, 2), &value);

status = viUnmapAddress(instr);

/* Close down the system */
status = viClose(instr);
status = viClose(defaultRM);
return 0;
}

```

## Visual Basic Example

```

Private Sub vbMain()

    Dim stat    As ViStatus
    Dim dfltRM  As ViSession
    Dim sesn    As ViSession
    Dim addr    As ViAddr
    Dim mSpace  As Integer
    Dim Value   As Integer

    Rem Open Default Resource Manager
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then

        Rem Error initializing VISA...exiting
        Exit Sub

    End If

    Rem Open communication with VXI Device at Logical Address 16
    Rem NOTE: For simplicity, we will not show error checking
    stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesn)

```

```

mSpace = VI_A16_SPACE

stat = viMapAddress(sesn, mSpace, 0, &H40, VI_FALSE, VI_NULL, addr)

viPeek16 sesn, addr, Value
Rem Access a different register by manipulating the pointer.
viPeek16 sesn, addr + 2, Value

stat = viUnmapAddress(sesn)

Rem Close down the system

stat = viClose(sesn)
stat = viClose(dfltRM)

End Sub

```

## Speed

In terms of the speed of developing your application, the HLA operations are much faster to implement and debug because of the simpler interface and the status information received after each access. For example, HLA operations encapsulate the mapping and unmapping of hardware windows, which means that you do not need to call `viMapAddress()` and `viUnmapAddress()` separately.

For speed of execution, the LLA operations perform faster when used for several random register I/O accesses in a single window. If you know that the next several accesses are within a single window, you can perform the mapping just once and then each of the accesses has minimal overhead.

The HLA operations will be slower because they must perform a map, access, and unmap within each call. Even if the window is correctly mapped for the access, the HLA call at the very least needs to perform some sort of check to determine if it needs to remap. Furthermore, because HLA operations encapsulate many status-checking capabilities not included in LLA operations, HLA operations have higher software overhead. For these reasons, HLA is slower than LLA in many cases.



**Note** For block transfers, the high-level `viMoveXX()` operations perform the

fastest.

## Ease of Use

HLA operations are easier to use because they encapsulate many status checking capabilities not included in LLA operations, which explains the higher software overhead and lower execution speed of HLA operations. HLA operations also encapsulate the mapping and unmapping of hardware windows, which means that you do not need to call `viMapAddress()` and `viUnmapAddress()` separately.

## Accessing Multiple Address Spaces

You can use LLA operations to access only the address space currently mapped. To access a different address space, you need to perform a remapping, which involves calling `viUnmapAddress()` and `viMapAddress()`. Therefore, LLA programming becomes more complex, without much of a performance increase, for accessing several address spaces concurrently. In these cases, the HLA operations are superior.

In addition, if you have several sessions to the same or different devices all performing register I/O, they must compete for the finite number of windows available. When using LLA operations, you must allocate the windows and always ensure that the program does not ask for more windows than are available. The HLA operations avoid this problem by restoring the window to the previous setting when they are done. Even if all windows are currently in use by LLA operations, you can still use HLA functions because they will save the state of the window, remap, access, and then restore the window. As a result, you can have an unlimited number of HLA windows.

## Shared Memory Operations



**Note** There are two distinct cases for using shared memory operations. In the first case, the local VXI controller exports general-purpose memory to the A24/A32 space. In the second case, remote VXI devices export memory into A24/A32 space. Unlike the first case, the memory exported to A24/A32 space may not be general purpose, so the VISA Shared Memory services do not control memory on remote VXI devices.

A common configuration in a VXI system is to export memory to either the A24 or A32 space. The local controller usually can export such memory. This memory can then be used to buffer the data going to or from the instruments in the system. However, a common problem is preventing multiple devices from using the same memory. In other words, a memory manager is needed on this memory to prevent corruption of the data.

The VISA Shared Memory operations—`viMemAlloc()` and `viMemFree()`—provide the memory management for a specific device, namely, the local controller. Since these operations are part of the INSTR resource, they are associated with a single VXI device. In addition, because a VXI device can export memory in either A24 or A32 space (but not both), the memory pool available to these operations is defined at startup. You can determine whether the memory resides in A24 or A32 space by querying the attribute `VI_ATTR_MEM_SPACE`.

## Shared Memory Sample Code

The following example shows how these shared memory operations work by incorporating them into the pointer manipulation example. Their main purpose is to allocate a block of memory from the pool that can then be accessed through the standard register-based access operations (high level or low level). The INSTR resource for this device ensures that no two sessions requesting memory receive overlapping blocks.

### C Example

```
#include "visa.h"

#define ADD_OFFSET (addr, offs) (((ViPByte)addr) + (offs))

int main(void)
{
    ViStatus      status;          /* For checking errors */
    ViSession     defaultRM,       /* Communication channels */
    ViAddr        self;           /* User pointer */
    ViBusAddress   address;        /* Shared memory offset */
    ViUInt16      offset;         /* Shared memory space */
    ViUInt16      addrSpace;      /* To store register value */
    value;                     /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
```



```

Logical Address 0 */
checking */
VI_NULL, VI_NULL,

VI_ATTR_MEM_SPACE, &addrSpace);

0x100, VI_FALSE,

the pointer. */

if (status < VI_SUCCESS) {
/* Error Initializing VISA...exiting */
return -1;
}
/* Open communication with VXI Device at

/* NOTE: For simplicity, we will not show error

status = viOpen(defaultRM, "VXI0::0::INSTR",

&self);

/* Allocate a portion of the device's memory */
status = viMemAlloc(self, 0x100, &offset);

/* Determine where the shared memory resides */
status = viGetAttribute(self,

status = viMapAddress(self, addrSpace, offset,

VI_NULL, &address);

viPeek16(self, address, &value);
/* Access a different register by manipulating

viPeek16(self, ADD_OFFSET(address, 2), &value);

status = viUnmapAddress(self);
status = viMemFree(self, offset);

/* Close down the system */
status = viClose(self);
status = viClose(defaultRM);
return 0;

}

```

## Visual Basic Example

```

Private Sub vbMain()

    Dim stat    As ViStatus
    Dim dfltRM  As ViSession
    Dim self    As ViSession
    Dim addr    As ViAddr

```

```

Dim offs      As Long
Dim mSpace    As Integer
Dim Value     As Integer

Rem Begin by initializing the system
stat = viOpenDefaultRM(dfltRM)
If (stat < VI_SUCCESS) Then
    Rem Error initializing VISA...exiting
    Exit Sub
End If

Rem Open communication with VXI Device at Logical Address 0
Rem NOTE: For simplicity, we will not show error checking
stat = viOpen(dfltRM, "VXI0::0::INSTR", VI_NULL, VI_NULL, self)

Rem Allocate a portion of the device's memory
stat = viMemAlloc(self, &H100, offs)

Rem Determine where the shared memory resides
stat = viGetAttribute(self, VI_ATTR_MEM_SPACE, mSpace)
stat = viMapAddress(self, mSpace, offs, &H100, VI_FALSE, VI_NULL, addr)
viPeek16 self, addr, Value
Rem Access a different register by manipulating the pointer.
viPeek16 self, addr + 2, Value

stat = viUnmapAddress(self)
stat = viMemFree(self, offs)

Rem Close down the system
stat = viClose(self)
stat = viClose(dfltRM)

End Sub

```

## Related concepts:

- [Manipulating the Pointer](#)

# VISA Events

VISA defines a common mechanism to notify an application when certain conditions occur. These conditions or occurrences are referred to as events. An event is a means of communication between a VISA resource and its applications. Typically, events occur because of a condition requiring the attention of applications.

The VISA event model provides the following two different ways for an application to receive event notification:

- The first method uses a queuing mechanism. You can use this method to place all of the occurrences of a specified event in a queue. The queuing mechanism is generally useful for noncritical events that do not need immediate servicing. [Queuing](#) describes this mechanism in detail.
- The other method is to have VISA invoke a function that the program specifies prior to enabling the event. This is known as a callback handler and is invoked on every occurrence of the specified event. The callback mechanism is useful when your application requires an immediate response. [Callbacks](#) describes this mechanism in detail.

The queuing and callback mechanisms are suitable for different programming styles. However, because these mechanisms work independently of each other, you can have them both enabled at the same time.

## Related concepts:

- [Queuing](#)
- [Callbacks](#)

## Supported Events

VISA defines the following generic and INSTR-specific event types.

Table 1. Generic and INSTR-Specific Event Types

| Event Type             | Description   | Resource Class(es), Other Notes  |
|------------------------|---|--|
| VI_EVENT_IO_COMPLETION | Notification that an asynchronous I/O operation has completed.                                | The I/O Completion event applies to all asynchronous operations, which for INSTR includes viReadAsync(), viWriteAsync(), and viMoveAsync(). For resource classes that do not support asynchronous operations, this event type is not applicable. |
| VI_EVENT_EXCEPTION     | Notification that an error condition (exception) has occurred during an operation invocation. | The exception event supports only the callback model. Refer to Exception Handling for more information about this event type.  |
| VI_EVENT_SERVICE_REQ   | Notification of a service request (SRQ) from the device.                                      | Supported for message based INSTR classes, including GPIB, VXI, and TCPIP.   |
| VI_EVENT_VXI_SIGP      | Notification of a VXIbus signal or VXIbus interrupt from the device.                          | Supported for VXI INSTR only.  |
| VI_EVENT_VXI_VME_INTR  | Notification of a VXIbus interrupt from the device.   | Supported for VXI INSTR only. This applies to both VXI and VME devices.  |
| VI_EVENT_TRIG          | Notification of a VXIbus trigger.   | Supported for VXI INSTR and VXI BACKPLANE only.  |
| VI_EVENT_PXI_INTR      | Notification of a PCI/PXI interrupt   | Supported for PXI INSTR only. Not supported on all platforms.  |

| Event Type          | Description  | Resource Class(es), Other Notes  |
|---------------------|--|--|
|                     | from the device.   |  |
| VI_EVENT_ASRL_BREAK | Notification that a break signal was received.                       | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac.   |
| VI_EVENT_ASRL_CTS   | Notification that the Clear To Send (CTS) line changed state.        | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac. If the CTS line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications. |
| VI_EVENT_ASRL_DCD   | Notification that the Data Carrier Detect (DCD) line changed state.  | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac. If the DCD line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications. |
| VI_EVENT_ASRL_DSR   | Notification that the Data Set Ready (DSR) line changed state.       | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac. If the DSR line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications. |
| VI_EVENT_ASRL_RI    | Notification that the Ring Indicator (RI) input signal was asserted. | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac.   |
| VI_EVENT_ASRL_CHAR  | Notification that at least one data byte has been received.          | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac. Each data character will not necessarily  |

| Event Type             | Description  | Resource Class(es), Other Notes   |
|------------------------|--|---|
|                        |  | result in an event notification.  |
| VI_EVENT_ASRL_TERMCHAR | Notification that the termination character has been received. | Supported for Serial INSTR only. This event is supported for all serial ports on Windows and LabVIEW RT, and ENET-Serial on all platforms. Except for ENET-Serial, it is not supported for serial ports on Linux or Mac. The actual termination character is specified by setting VI_ATTR_TERMCHAR prior to enabling this event. For this event, the setting of VI_ATTR_TERMCHAR_EN is ignored. |

To learn about other event types defined for other resource classes, refer to Interface-Specific Information or the appropriate event topics.

VISA events use a list of attributes to maintain information associated with the event. You can access the event attributes using the `viGetAttribute()` operation, just as for the session and resource attributes. Remember to use the `eventContext` as the first parameter, rather than the I/O session.

All VISA events support the generic event attribute `VI_ATTR_EVENT_TYPE`. This attribute returns the identifier of the event type. In addition to this attribute, individual events may define attributes to hold additional event information. The events listed below define the accompanying additional attributes; the other event types do not define any additional attributes.

- `VI_EVENT_IO_COMPLETION` defines, among other attributes, `VI_ATTR_STATUS` and `VI_ATTR_RET_COUNT/VI_ATTR_RET_COUNT_32/VI_ATTR_RET_COUNT_64`, which provide information about how the asynchronous I/O operation completed.
- `VI_EVENT_VXI_SIGP` defines `VI_ATTR_SIGP_STATUS_ID`, which contains the 16-bit Status/ID value retrieved during the interrupt or from the Signal register.
- `VI_EVENT_VXI_VME_INTR` defines `VI_ATTR_RECV_INTR_LEVEL` and `VI_ATTR_INTR_STATUS_ID`, which provide the interrupt level and 32-bit interrupt Status/ID value, respectively.
- `VI_EVENT_TRIG` defines `VI_ATTR_RECV_TRIG_ID`, which provides the trigger line on which the trigger was received.
- `VI_EVENT_EXCEPTION` defines `VI_ATTR_STATUS` and `VI_ATTR_OPER_NAME`, which provide information about what error was generated and which operation generated it, respectively.

All the attributes VISA events support are read-only attributes; a user application cannot modify their values. Refer to the appropriate event topics for detailed information on the specific events.

### Related concepts:

- [Exception Handling](#)

## Enabling and Disabling Events

Before a session can use either the VISA callback or queuing mechanism, you need to enable the session to sense events. You use the `viEnableEvent()` operation to enable an event type using either of the mechanisms. For example, to enable the `VI_EVENT_SERVICE_REQ` event for queuing, use the following code:

```
status = viEnableEvent(instr,VI_EVENT_SERVICE_REQ,VI_QUEUE,VI_NULL);
```



**Note** VISA currently allows both queuing and callbacks to be enabled for the same event type on the same session. You can do this in one call by bitwise ORing the mechanisms together (`VI_QUEUE|VI_HNDLR`), or you can do this in two separate calls to `viEnableEvent()`. The two mechanisms operate independently of each other. However, using both mechanisms for the same event type on the same session is usually unnecessary and is difficult to debug. Therefore, this is highly discouraged.

Use `viDisableEvent()` to stop a session from receiving events of a specified type. You can specify the mechanism for which you are disabling, although it is more convenient to use `VI_ALL_MECH` to disable the event type for all mechanisms. For example, to disable the `VI_EVENT_SERVICE_REQ` event regardless of the mechanism for which it was enabled, use the following code:

```
status = viDisableEvent(instr,VI_EVENT_SERVICE_REQ,VI_ALL_MECH);
```

The `viEnableEvent()` operation also automatically enables the hardware, if necessary for detecting the event. The hardware is enabled when the first call to `viEnableEvent()` for the event is made from any of the sessions currently active. Similarly,

`viDisableEvent()` disables the hardware when the last enabled session disables itself for the event.

## Queuing

The queuing mechanism in VISA gives an application the flexibility to receive events only when it requests them. An application uses the `viWaitOnEvent()` operation to retrieve the event information. However, in addition to retrieving events from the queue, you can also use `viWaitOnEvent()` in your application to halt the current execution and wait for the event to arrive. Both of these cases are discussed in this section.

The event queuing process requires that you first enable the session to sense the particular event type. When enabled, the session can automatically queue the event occurrences as they happen. A session can later dequeue these events using the `viWaitOnEvent()` operation. You can set the timeout to `VI_TMO_IMMEDIATE` if you want your application to check if any event of the specified event type exists in the queue.



**Note** Each session has a queue for each of the possible events that can occur. This means that each queue is per session and per event type.

An application can also use `viWaitOnEvent()` to wait for events if none currently exists in the queue. When you select a non-zero timeout value (something other than `VI_TMO_IMMEDIATE`), the operation retrieves the specified event if it exists in the queue and returns immediately. Otherwise, the application waits until the specified event occurs or until the timeout expires, whichever occurs first. When an event arrives and causes `viWaitOnEvent()` to return, the event is not queued for the session on which the wait operation was invoked. However, if any other session is currently enabled for queuing, the event is placed on the queue for that session.

You can use `viDisableEvent()` to disable event queuing on a session, as discussed in the previous section. After calling `viDisableEvent()`, no further event occurrences are queued on that session, but event occurrences that were already in the event queue are retained. Your application can use `viWaitOnEvent()` to dequeue these retained events in the same manner as previously described. The wait operation does not need



to have events enabled to work; however, the session must be enabled to detect new events. An application can explicitly clear (flush) the event queue with the `viDiscardEvents()` operation.

The event queues in VISA do not dynamically grow as new events arrive. The default queue length is 50, but you can change the size of a queue by using the `VI_ATTR_MAX_QUEUE_LENGTH` template attribute. This attribute specifies the maximum number of events that can be placed in a queue.



**Note** If the event queue is full and a new event arrives, the new event is discarded.

VISA does not let you dynamically configure queue lengths. That is, you can only modify the queue length on a given session before the first invocation of the `viEnableEvent()` operation, as shown in the following code segment.

```
status = viSetAttribute(instr, VI_ATTR_MAX_QUEUE_LENGTH, 10);  
status = viEnableEvent(instr, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL);
```

See the handling events example for an example of handling events via the queue mechanism.

### Related concepts:

- [Example of Handling Events](#)

## Callbacks

The VISA event model also allows applications to install functions that can be called back when a particular event type is received. You need to install a handler before enabling a session to sense events through the callback mechanism. Refer to `userHandle` Parameter for more information. The procedure works as follows:

1. Use the `viInstallHandler()` operation to install handlers to receive events.
2. Use the `viEnableEvent()` operation to enable the session for the callback mechanism as described in [Enabling and Disabling Events](#).

3. The VISA driver invokes the handler on every occurrence of the specified event.
4. VISA provides the event object in the `eventContext` parameter of `viEventHandler()`. The **event context** is like a data structure, and contains information about the specific occurrence of the event. Refer to The Life of the Event Context for more information on event context.

You can now have multiple handlers per session in the current revision of VISA. If you have multiple handlers installed for the same event type on the same session, each handler is invoked on every occurrence of that event type. The handlers are invoked in reverse order of installation; that is, in Last In First Out (LIFO) order. For a given handler to prevent other handlers on the same session from being executed, it should return the value `VI_SUCCESS_NCHAIN` rather than `VI_SUCCESS`. This does not affect the invocation of event handlers on other sessions or in other processes.

### Related concepts:

- [Enabling and Disabling Events](#)

## Callback Modes

VISA gives you the choice of two different modes for using the callback mechanism. You can use either direct callbacks or suspended callbacks. You can have only one of these callback modes enabled at any one time.

To use the direct callback mode, specify `VI_HNDLR` in the `mechanism` parameter. In this mode, VISA invokes the callback routine at the time the event occurs.

To use the suspended callback mode, specify `VI_SUSPEND_HNDLR` in the `mechanism` parameter. In this mode, VISA does not invoke the callback routine at the time of event occurrence; instead, the events are placed on a suspended handler queue. This queue is similar to the queue used by the queuing mechanism except that you cannot access it directly. You can obtain the events on the queue only by re-enabling the session for callbacks. You can flush the queue with `viDiscardEvents()`.

For example, the following code segment shows how you can halt the arrival of events while you perform some critical operations that would conflict with code in the

callback handler. Notice that no events are lost while this code executes, because they are stored on a queue.

```
status = viEnableEvent(instr, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);
.
.
.
status = viEnableEvent(instr, VI_EVENT_SERVICE_REQ, VI_SUSPEND_HNDLR, VI_NULL);

/*Perform code that must not be interrupted by a callback. */

status = viEnableEvent(instr, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);
```

When you switch the event mechanism from `VI_HNDLR` to `VI_SUSPEND_HNDLR`, the VISA driver can still detect the events. For example, VXI interrupts still generate a local interrupt on the controller and VISA handles these interrupts. However, the event VISA generates for the VXI interrupt is now placed on the handler queue rather than passed to the application. When the critical section completes, switching the mechanism from `VI_SUSPEND_HNDLR` back to `VI_HNDLR` causes VISA to call the application's callback functions whenever it detects a new event as well as for every event waiting on the handler queue.

## Independent Queues

As stated previously, the callback and the queuing mechanisms operate totally independently of each other, so VISA keeps the information for event occurrences separately for both mechanisms. Therefore, VISA maintains the suspended handler queue separately from the event queue used for the queuing mechanism. The `VI_ATTR_MAX_QUEUE_LENGTH` attribute mentioned in Queuing applies to the suspended handler queue as well as to the queue for the queuing mechanism. However, because these queues are separate, if one of the queues reaches the predefined limit for storing event occurrences, it does not directly affect the other mechanism.

### Related concepts:

- [Queuing](#)

## The userHandle Parameter

When using `viInstallHandler()` to install handlers for the callback mechanism, your application can use the `userHandle` parameter to supply a reference to any application-defined value. This reference is passed back to the application as the `userHandle` parameter to the callback routine during handler invocation. By supplying different values for this parameter, applications can install the same handler with different application-defined contexts.

For example, applications often need information that was received in the callback to be available for the main program. In the past, this has been done through global variables. In VISA, `userHandle` gives the application more modularity than is possible with global variables. In this case, the application can allocate a data structure to hold information locally. When it installs the callback handler, it can pass the reference to this data structure to the callback handler via the `userHandle`. This means that the handler can store the information in the local data structure rather than a global data structure.

For another example, consider an application that installs a handler with a fixed value of `0x1` for the `userHandle` parameter. It can install the same handler with a different value (for example, `0x2`) for the same event type on another session. However, installations of the same handler are different from one another. Both handlers are invoked when the event of the given type occurs, but in one invocation the value passed to `userHandle` is `0x1`, and in the other it is `0x2`. As a result, you can uniquely identify VISA event handlers by a combination of the handler address and user context pair.

This structure also is important when the application attempts to remove the handler. The operation `viUninstallHandler()` requires not only the handler's address but also the `userHandle` value to correctly identify which handler to remove.

## Queuing and Callback Mechanism Sample Code

This example demonstrates the use of both the queuing and callback mechanisms in event handling. In the program, a message is sent to a GPIB device telling it to read some data. When the data collection is complete, the device asserts SRQ, informing the program that it can now read data. After reading the device's status byte, the

handler begins to read asynchronously using a buffer of information that the main program passes to it.

## C Example

```
#include "visa.h"
#include <stdlib.h>

#define MAX_CNT 1024

/* This function is to be called when an SRQ event occurs */
/* Here, an SRQ event indicates the device has data ready */
ViStatus _VI_FUNCH myCallback(ViSession vi, ViEventType etype, ViEvent
eventContext, ViAddr userHandle)
{
    ViJobId    jobID;
    ViStatus    status;
    ViUInt16    stb;

    status = viReadSTB(vi, &stb);
    status = viReadAsync(vi, (ViBuf)userHandle, MAX_CNT, &jobID);
    return VI_SUCCESS;
}

int main(void)
{
    ViStatus    status;
    ViSession    defaultRM, gpibSesn;
    ViBuf        bufferHandle;
    ViUInt32    retCount;
    ViEventType    etype;
    ViEvent        eventContext;

    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error initializing VISA...exiting */
        return -1;
    }

    /* Open communication with GPIB device at primary address 2 */
```

```

status = viOpen(defaultRM, "GPIB0::2::INSTR", VI_NULL, VI_NULL, &gpibSesn);

/* Allocate memory for buffer */
/* In addition, allocate space for the ASCII NULL character */
bufferHandle = (ViBuf)malloc(MAX_CNT+1);

/* Tell the driver what function to call on an event */
status = viInstallHandler(gpibSesn, VI_EVENT_SERVICE_REQ, myCallback,
bufferHandle);

/* Enable the driver to detect events */
status = viEnableEvent(gpibSesn, VI_EVENT_SERVICE_REQ, VI_HNDLR, VI_NULL);
status = viEnableEvent(gpibSesn, VI_EVENT_IO_COMPLETION, VI_QUEUE, VI_NULL);

/* Tell the device to begin acquiring a waveform */
status = viWrite(gpibSesn, "E0x51; W1", 9, &retCount);

/* The device asserts SRQ when the waveform is ready */
/* The callback begins reading the data */
/* After the data is read, an I/O completion event occurs */

status = viWaitOnEvent(gpibSesn, VI_EVENT_IO_COMPLETION, 20000, &etype,
&eventContext);
if (status < VI_SUCCESS) {

    /* Waveform not received...exiting */
    free(bufferHandle);
    viClose(defaultRM);
    return -1;
}
/* Your code should process the waveform data */

/* Close the event context */
viClose(eventContext);

/* Stop listening for events */
status = viDisableEvent(gpibSesn, VI_ALL_ENABLED_EVENTS, VI_ALL_MECH);
status = viUninstallHandler(gpibSesn, VI_EVENT_SERVICE_REQ,
myCallback,bufferHandle);

/* Close down the system */
free(bufferHandle);
status = viClose(gpibSesn);
status = viClose(defaultRM);
return 0;

```

```
}
```

## Visual Basic Example

Visual Basic does not support callback handlers, so currently the only way to handle events is through `viWaitOnEvent()`. Because Visual Basic does not support asynchronous operations either, this example uses the `viRead()` call instead of the `viReadAsync()` call.

```
Private Sub vbMain()

    Const MAX_CNT = 1024

    Dim stat           As ViStatus
    Dim dfltRM         As ViSession
    Dim sesn           As ViSession
    Dim bufferHandle    As String
    Dim retCount       As Long
    Dim etype          As ViEventType
    Dim event          As ViEvent
    Dim stb            As Integer

    Rem Begin by initializing the system
    Rem NOTE: For simplicity, we will not show error checking
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then

        Rem Error initializing VISA...exiting
        Exit Sub

    End If

    Rem Open communication with GPIB device at primary address 2
    stat = viOpen(dfltRM, "GPIB0::2::INSTR", VI_NULL, VI_NULL, sesn)

    Rem Allocate memory for buffer
    Rem In addition, allocate space for the ASCII NULL character
    bufferHandler = Space$(MAX_CNT + 1)

    Rem Enable the driver to detect events
    stat = viEnableEvent(sesn, VI_EVENT_SERVICE_REQ, VI_QUEUE, VI_NULL)

    Rem Tell the device to begin acquiring a waveform
```

```

stat = viWrite(sesn, "E0x51; W1", 9, retCount)

Rem The device asserts SRQ when the waveform is ready
stat = viWaitOnEvent(sesn, VI_EVENT_SERVICE_REQ, 20000, etype, event)
If (stat < VI_SUCCESS) Then

    Rem Waveform not received...exiting
    stat = viClose (dfltRM)
    Exit Sub

End If
stat = viReadSTB (sesn, stb)

Rem Read the data
stat = viRead(sesn, bufferHandle, MAX_CNT, retCount)
Rem Your code should process the waveform data

Rem Close the event context
stat = viClose (event)

Rem Stop listening for events
stat = viDisableEvent(sesn, VI_ALL_ENABLED_EVENTS, VI_ALL_MECH)

Rem Close down the system
stat = viClose(sesn)
stat = viClose(dfltRM)

End Sub

```

## Event Context

The event context that the VISA driver generates when an event occurs is a data object that contains the information about the event. Because it is more than just a simple variable, memory allocation and deallocation becomes important.

### Event Context with the Queuing Mechanism

When you use the queuing mechanism, the event context is returned when you call `viWaitOnEvent()`. The VISA driver has created this data structure, but it cannot destroy it until you tell it to. For this reason, in VISA you call `viClose()` on the event context so the driver can free the memory for you. Always remember to call `viClose()` when you are done with the event.



If you know the type of event you are receiving, and the event does not provide any useful information to your application other than whether it actually occurred, you can pass `VI_NULL` as the `outEventType` and `eventContext` parameters as shown in the following example:

```
status = viWaitOnEvent(gpibSesn, VI_EVENT_SERVICE_REQ, 5000, VI_NULL, VI_NULL);
```

In this case, VISA automatically closes the event data structure rather than returning it to you. Calling `viClose()` on the event context is therefore both unnecessary and incorrect because VISA would not have returned the event context to you.

## Event Context with the Callback Mechanism

In the case of callbacks, the event is passed to you in a function, so the VISA driver has a chance to destroy it when the function ends. This has two important repercussions. First, you do not need to call `viClose()` on the event inside the callback function. Indeed, calling this operation on the event could lead to serious problems because VISA will access the event (to close it) when your callback returns. Secondly, the event itself has a life only as long as the callback function is executing. Therefore, if you want to keep any information about the event after the callback function, you should use `viGetAttribute()` to retrieve the information for storage. Any references to the event itself becomes invalid when the callback function ends.

## Exception Handling

By using the VISA event `VI_EVENT_EXCEPTION`, you can have one point in your code that traps all errors and handles them appropriately. This means that after you install and enable your VISA exception handler, you do not have to check the return status from each operation, which makes the code easier to read and maintain. How an application handles error codes is specific to both the device and the application. For one application, an error could mean different things from different devices, and might even be ignored under certain circumstances; for another, any error could always be fatal.

For an application that needs to treat all errors as fatal, one possible use for this event type would be to print out a debug message and then exit the application. Because the method of installing the handler and then enabling the event has already been

covered, the following code segment shows only the handler itself:

```
ViStatus _VI_FUNCH myEventHandler (ViSession vi, ViEventType etype, ViEvent
eventContext, ViAddr uHandle)
{
    ViChar rsrcName[256], operName[256];
    ViStatus stat;
    ViSession rm;
    if (etype == VI_EVENT_EXCEPTION) {
        viGetAttribute(vi, VI_ATTR_RSRC_NAME, rsrcName);
        viGetAttribute(eventContext, VI_ATTR_OPER_NAME, operName);
        viGetAttribute(eventContext, VI_ATTR_STATUS, &stat);
        printf(
            "Session 0x%08lX to resource %s caused error 0x%08lX in operation %s.\n",
            vi, rsrcName, stat, operName);

        /* Use this code only if you will not return control to VISA */
        viGetAttribute(vi, VI_ATTR_RM_SESSION, &rm);
        viClose(eventContext);
        viClose(vi);
        viClose(rm);
        exit(-1); /* exit the application immediately */
    }
    /* code for other event types */
    return VI_SUCCESS;
}
```

If you wanted just to print a message, you would leave out the code that closes the objects and exits. Notice that in this code segment, the event object is closed inside of the callback, even though we just recommended in the previous section that you not do this! The reason that we do it here is that the code will never return control to VISA—calling `exit()` will return control to the operation system instead. This is the only case where you should ever invoke `viClose()` within a callback.

Another (more advanced) use of this event type is for throwing C++ exceptions. Because VISA exception event handlers are invoked in the context of the same thread in which the error condition occurs, you can safely throw a C++ exception from the VISA handler. Like the example above, you would invoke `viClose()` on the exception event (but you would probably not close the actual session or its resource manager session). You would also need to include the information about the VISA exception (for example, the status code) in your own exception class (of the type that you throw), since this will not be available once the VISA event is closed.

Throwing C++ exceptions introduces several issues to consider. First, if you have mixed C and C++ code in your application, this could introduce memory leaks in cases where C functions allocate local memory on the heap rather than the stack. Second, if you use asynchronous operations, an exception is thrown only if the error occurs before the operation is posted (for example, if the error generated is `VI_ERROR_QUEUE_ERROR`). If the error occurs during the operation itself, the status is returned as part of the `VI_EVENT_IO_COMPLETION` event. This is important because that event may occur in a separate thread, due to the nature of asynchronous I/O. Therefore, you should not use asynchronous operations if you want to throw C++ exceptions from your handler.

# VISA Locks

VISA introduces locks for access control of resources. In VISA, applications can open multiple sessions to a resource simultaneously and can access the resource through these different sessions concurrently. In some cases, applications accessing a resource must restrict other sessions from accessing that resource. For example, an application may need to execute a write and a read operation as a single step so that no other operations intervene between the write and read operations. The application can lock the resource before invoking the write operation and unlock it after the read operation, to execute them as a single step. VISA defines a locking mechanism to restrict accesses to resources for such special circumstances.

The VISA locking mechanism enforces arbitration of accesses to resources on an individual basis. If a session locks a resource, operations invoked by other sessions are serviced or returned with a locking error, depending on the operation and the type of lock used.

## Lock Types

VISA defines two different types, or modes, of locks: **exclusive** and **shared** locks, which are denoted by `VI_EXCLUSIVE_LOCK` and `VI_SHARED_LOCK`, respectively. `viLock()` is used to acquire a lock on a resource, and `viUnlock()` is used to release the lock.

If a session has an exclusive lock, other sessions cannot modify global attributes or invoke operations, but can still get attributes and set local attributes. If the session has a shared lock, other sessions that have shared locks can also modify global attributes and invoke operations.

Regardless of which type of lock a session has, if the session is closed without first being unlocked, VISA automatically performs a `viUnlock()` on that session.

## Lock Sharing

The locking mechanism in VISA is session based, not thread based. Therefore, if

multiple threads share the same session, they have the same privileges for accessing the resource. VISA locks will not provide mutual exclusion in this scenario. However, some applications might have separate sessions to a resource for these multiple threads, and might require that all the sessions in the application have the same privileges as the session that locked the resource. In other cases, there might be a need to share locks among sessions in different applications. Essentially, sessions that have a lock to a resource may share the lock with certain sessions, and exclude access from other sessions.

This section discusses the mechanism that makes it possible to share locks. VISA defines a lock type—`VI_SHARED_LOCK`—that gives exclusive access privileges to a session, along with the capability to share these exclusive privileges at the discretion of the original session. When locking sessions with a shared lock, the locking session gains an access key. The session can then share this lock with any other session by passing the access key. VISA allows user applications to specify an access key to be used for lock sharing, or VISA can generate the access key for an application.

If the application chooses to specify the `accessKey`, other sessions that want access to the resource must choose the same unique `accessKey` for locking the resource. Otherwise, when VISA generates the `accessKey`, the session that gained the shared lock should make the `accessKey` available to other sessions for sharing access to the locked resource. Before the other sessions can access the locked resource, they must acquire the lock using the same access key in the `accessKey` parameter of the `viLock()` operation. Invoking `viLock()` with the same access key will register the new session with the same access privileges as the original session. All sessions that share a resource should synchronize their accesses to maintain a consistent state of the resource. The following code is an example of obtaining a shared lock with a requested name:

```
status = viLock(instr, VI_SHARED_LOCK, 15000, "MyLockName", accessKey);
```

This example attempts to acquire a shared lock with "MyLockName" as the `requestedKey` and a timeout of 15 s. If the call is successful, `accessKey` will contain "MyLockName". If you want to have VISA generate a key, simply pass `VI_NULL` in place of "MyLockName" and VISA will return a unique key in `accessKey` that other sessions can use for locking the resource.

## Acquiring an Exclusive Lock While Owning a Shared Lock

When multiple sessions have acquired a shared lock, VISA allows one of the sessions to acquire an exclusive lock as well as the shared lock it is holding. That is, a session holding a shared lock can also acquire an exclusive lock using the `viLock()` operation. The session holding both the exclusive and shared lock has the same access privileges it had when it was holding only the shared lock. However, the exclusive lock precludes other sessions holding the shared lock from accessing the locked resource. When the session holding the exclusive lock unlocks the resource using the `viUnlock()` operation, all the sessions (including the one that acquired the exclusive lock) again have all the access privileges associated with the shared lock. This circumstance is useful when you need to synchronize multiple sessions holding a shared lock. A session holding an exclusive and shared lock can also be useful when one of the sessions needs to execute in a critical section.

## Nested Locks

VISA supports nested locking. That is, a session can lock the same resource multiple times (for the same lock type). Unlocking the resource requires an equal number of invocations of the `viUnlock()` operation. Each session maintains a separate lock count for each type of locks. Repeated invocations of the `viLock()` operation for the same session increase the appropriate lock count, depending on the type of lock requested. In the case of shared locks, nesting `viLock()` calls return with the same `accessKey` every time. In the case of exclusive locks, `viLock()` does not return an `accessKey`, regardless of whether it is nested. For each invocation of `viUnlock()`, the lock count is decremented. VISA unlocks a resource only when the lock count equals 0.

## Locking Sample Code

This example uses a shared lock because two sessions are opened for performing trigger operations. The first session receives triggers and the second session sources triggers. A shared lock is needed because an exclusive lock would prohibit the other session from accessing the same resource. If `viWaitOnEvent()` fails, this example performs a `viClose()` on the resource manager without unlocking or closing the sessions. When the resource manager session closes, all sessions that were opened using it automatically close as well. Likewise, remember that closing a session that has

any lock results in automatically releasing its lock(s).

## C Example

```
#include "visa.h"
#define MAX_COUNT 128
int main(void)
{
    ViStatus      status;                /* For checking errors */
    ViSession     defaultRM;             /* Communication channels */
    ViSession     instrIN,               /* Communication channels */
    ViChar        instrOUT;              /* Access key for lock */
    ViByte        accKey[VI_FIND_BUFLen]; /* To store device data */
    ViEventType    buf[MAX_COUNT];       /* To identify event */
    ViEvent        etype;                 /* To hold event info */
    ViUInt32       event;                 /* To hold byte count */
    retCount;

    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error Initializing VISA...exiting */
        return -1;
    }
    /* Open communications with VXI Device at Logical Addr 16 */
    status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &instrIN);
    status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &instrOUT);

    /* We open two sessions to the same device */
    /* One session is used to assert triggers on TTL channel 4 */
    /* The second is used to receive triggers on TTL channel 5 */

    /* Lock first session as shared, have VISA generate the key */
    /* Then lock the second session with the same access key */

    status = viLock(instrIN, VI_SHARED_LOCK, 5000, VI_NULL, accKey);
    status = viLock(instrOUT, VI_SHARED_LOCK, VI_TMO_IMMEDIATE, accKey, accKey);

    /* Set trigger channel for sessions */
    status = viSetAttribute(instrIN, VI_ATTR_TRIG_ID, VI_TRIG_TTL5);
    status = viSetAttribute(instrOUT, VI_ATTR_TRIG_ID, VI_TRIG_TTL4);

    /* Enable input session for trigger events */
    status = viEnableEvent(instrIN, VI_EVENT_TRIG, VI_QUEUE, VI_NULL);

    /* Assert trigger to tell device to start sampling */
}
```

```

    status = viAssertTrigger(instrOUT, VI_TRIG_PROT_DEFAULT);

    /* Device will respond with a trigger when data is ready */
    if ((status = viWaitOnEvent(instrIN, VI_EVENT_TRIG, 20000, &etype, &event)) <
VI_SUCCESS) {
        viClose(defaultRM);
        return -1;
    }
    /* Close the event */
    status = viClose(event);

    /* Read data from the device */
    status = viRead(instrIN, buf, MAX_COUNT, &retCount);

    /* Your code should process the data */

    /* Unlock the sessions */
    status = viUnlock(instrIN);
    status = viUnlock(instrOUT);

    /* Close down the system */
    status = viClose(instrIN);
    status = viClose(instrOUT);
    status = viClose(defaultRM);
    return 0;
}

```

## Visual Basic Example

```

Private Sub vbMain()
Const MAX_COUNT = 128

Dim stat      As ViStatus      'For checking errors
Dim dfltRM    As ViSession    'Communication channels
Dim sesnIN    As ViSession    'Communication channels
Dim sesnOUT   As ViSession    'Communication channels
Dim aKey      As String * VI_FIND_BUFLen 'Access key for lock
Dim buf       As String * MAX_COUNT    'To store device data
Dim etype     As ViEventType  'To identify event
Dim event     As ViEvent      'To hold event info
Dim retCount  As Long         'To hold byte count

    Rem Begin by initializing the system
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then

```



```

Rem Error initializing VISA...exiting
Exit Sub

End If

Rem Open communications with VXI Device at Logical Addr 16
stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesnIN)
stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesnOUT)

Rem We open two sessions to the same device
Rem One session is used to assert triggers on TTL channel 4
Rem The second is used to receive triggers on TTL channel 5

Rem Lock first session as shared, have VISA generate the key
Rem Then lock the second session with the same access key
stat = viLock(sesnIN, VI_SHARED_LOCK, 5000, "", aKey)
stat = viLock(sesnOUT, VI_SHARED_LOCK, VI_TMO_IMMEDIATE, aKey, aKey)

Rem Set trigger channel for sessions
stat = viSetAttribute(sesnIN, VI_ATTR_TRIG_ID, VI_TRIG_TTL5)
stat = viSetAttribute(sesnOUT, VI_ATTR_TRIG_ID, VI_TRIG_TTL4)

Rem Enable input session for trigger events
stat = viEnableEvent(sesnIN, VI_EVENT_TRIG, VI_QUEUE, VI_NULL)

Rem Assert trigger to tell device to start sampling
stat = viAssertTrigger(sesnOUT, VI_TRIG_PROT_DEFAULT)

Rem Device will respond with a trigger when data is ready
stat = viWaitOnEvent(sesnIN, VI_EVENT_TRIG, 20000, etype, event)
If (stat < VI_SUCCESS) Then

    stat = viClose (dfltRM)
    Exit Sub

End If

Rem Close the event
stat = viClose(event)

Rem Read data from the device
stat = viRead(sesnIN, buf, MAX_COUNT, retCount)

Rem Your code should process the data

```

```
Rem Unlock the sessions
stat = viUnlock(sesnIN)
stat = viUnlock(sesnOUT)

Rem Close down the system
stat = viClose(sesnIN)
stat = viClose(sesnOUT)
stat = viClose(dfltRM)
```

```
End Sub
```

# Creating a .NET Application without Measurement Studio

With the Microsoft .NET Framework version 1.1 or later, you can use NI-VISA to create applications using Visual C# and Visual Basic .NET without Measurement Studio. You need Microsoft Visual Studio .NET 2005 or 2003 for the API documentation to be installed.

The installed documentation contains the NI-VISA API overview and function reference. This help is fully integrated into the Visual Studio .NET documentation. To view the VISA .NET documentation, go to **Start » Programs » National Instruments » VISA » NI-VISA .NET Framework 1.1 Help**. Expand **NI Measurement Studio Help » NI Measurement Studio .NET Class Library » Reference » NationalInstruments.VisaNS** to view the function reference. Expand **NI Measurement Studio Help » NI Measurement Studio .NET Class Library » Using the Measurement Studio .NET Class Libraries » Using the Measurement Studio VisaNS .NET Library** to view conceptual topics for using NI-VISA with Visual C# and Visual Basic .NET.

To get to the same help topics from within Visual Studio .NET 2003, go to **Help » Contents**. Select **Measurement Studio** from the **Filtered By** drop-down list and follow the previous instructions.

# NI-VISA Utilities

NI-VISA provides the following utilities.

## NI I/O Trace: Debugging Tool

NI I/O Trace tracks the calls your application makes to National Instruments test and measurement (T&M) drivers, including NI-VXI, NI-VISA, and NI-488.2.

NI I/O Trace highlights functions that return errors, so you can quickly determine which functions failed during your development. NI I/O Trace can also log your program's calls to these drivers into a file so you can check them for errors at your convenience.

## Interactive Control of VISA

NI-VISA comes with a utility called VISA Interactive Control (VISAIC) on Windows and Linux. This utility gives you access to all VISA functionality interactively in an easy-to-use graphical environment. It is a convenient starting point for program development and learning about VISA.

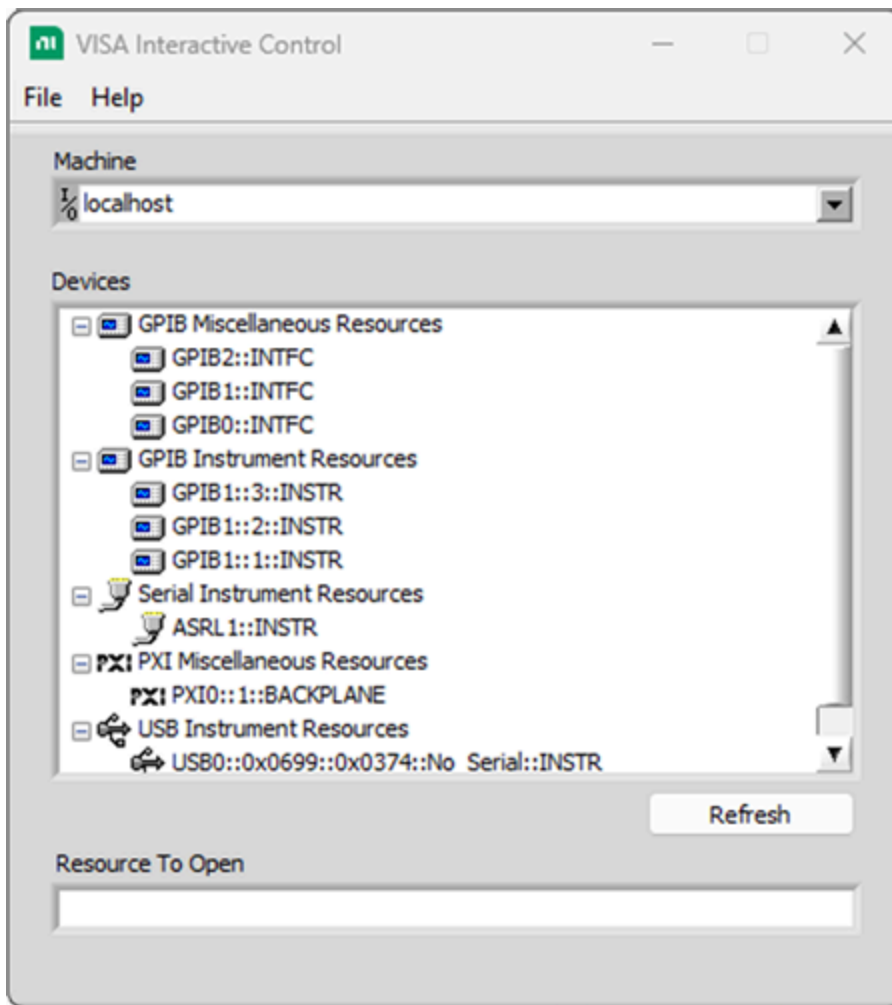


**Note** To launch VISAIC on Windows, select **Start » National Instruments » VISA » VISA Interactive Control**.



**Note** On Linux, VISAIC is called NIvisaic.

When VISAIC runs, it automatically finds all of the available resources in the system and lists the instrument descriptors for each of these resources under the appropriate resource type. This information is displayed in the **Devices** listbox .

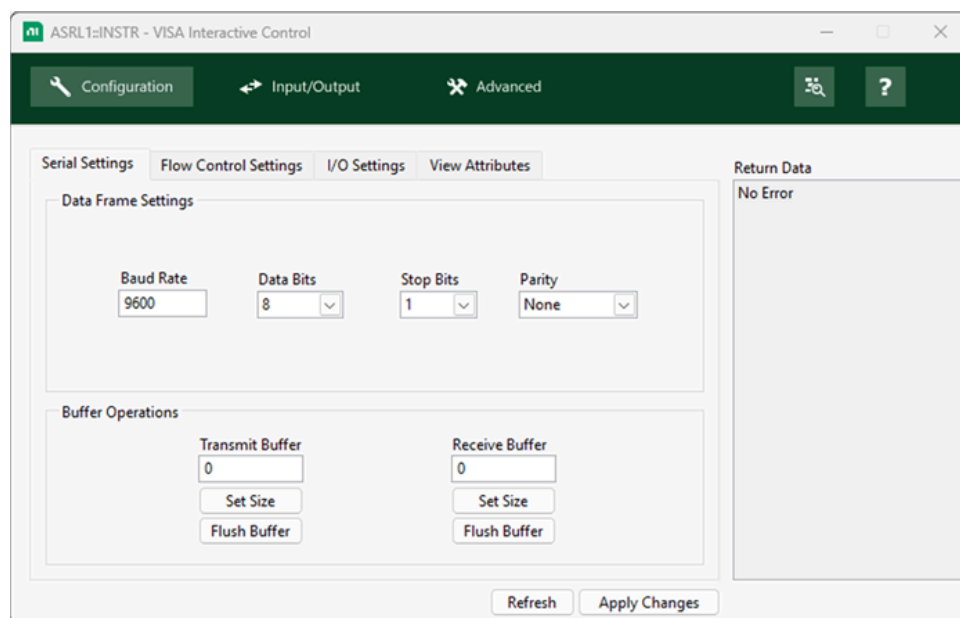


## VISAIC Window

The **Resource to Open** control in VISAIC is a string control for using VISA expressions to find specific instruments. Refer to `viFindRsrc` for information about creating custom expressions. You also can select a resource from the **Devices** listbox to populate the **Resource to Open** string control.

Double-clicking on any of the instrument descriptors shown in the VISAIC window opens a session to that instrument. Opening a session to the instrument launches a window with a series of tabs for interactively running VISA commands. You also can click the **Open VISA Test Panel** button in Measurement & Automation Explorer (MAX) to open a session to the instrument.

When a session to a resource is opened interactively, a window similar to the following appears.



The VISAIC Test Panel includes the following pages:

- **Configuration**—Contains all of the attributes that can be configured or set for the opened resource. The attributes are grouped together by category and separated into tabs. Open the **View Attributes** tab if you want to view all of the attributes for the selected resource.
- **Input/Output**—Contains operations for message-based and register-based resources. The Basic I/O tab is visible only for message-based instruments, while the Register I/O tab is visible only for register-based instruments. These tabs are visible only if they apply to the selected bus or instrument type.



**Note** The Input/Output page is disabled for backplane resources.

- **Advanced**—Contains all of the operations that deal with locks, triggers, and events. This page also contains low-level register I/O operations when applicable to the selected resource.

## Measurement & Automation Explorer (MAX)

Measurement & Automation Explorer (MAX) provides access to all NI DAQ, GPIB, IMAQ, IVI, Motion, VISA, and VXI devices. With MAX, you can configure NI hardware and software, add new channels, interfaces, and virtual instruments, execute system diagnostics, and view the devices and instruments connected to your system. Installs

automatically with NI-VISA version 2.5 or later or NI-VXI version 3.0 or later. Available only for Win32-based operating systems.

## Hardware Configuration Utility (HWCU)

Hardware Configuration Utility is software for test engineers and technicians who need to visualize and configure hardware in their systems. Use Hardware Configuration Utility to view, configure connected NI hardware and manage NI-VISA security credentials.

## visaconf

visaconf is the VISA configuration utility for Linux and Mac OS X.

## NI-VISA Driver Wizard Overview

To make your PXI/PCI or USB device visible to NI-VISA applications, the operating system (OS) must know to associate your hardware with the NI-VISA driver. This association is accomplished on Microsoft Windows operating systems using a Setup Information file (`.inf` file).

The NI-VISA Driver Wizard generates one `.inf` file for your PXI/PCI device for use on all supported operating systems. For a USB device, the wizard generates two `.inf` files, one for Windows XP/Server 2003 R2, the other for Windows 7 SP 1 and above. Using the wizard for a USB device is not necessary for use on Linux or Mac OS X. At this time, the list of supported operating systems includes Windows 10/8.1/7 SP1/Vista/XP/Server 2008 R2/Server 2003 R2, LabVIEW RT, Linux, and Mac OS X. The `.inf` file created by the wizard can then be distributed with an instrument driver distribution kit.

# Programming GPIB Devices in VISA

VISA supports programming IEEE 488.1 and IEEE 488.2 devices, and includes complete device-level and board-level functionality.

For novice GPIB users, the VISA API presents a simple interface for device communication. Most GPIB devices allow you to set a primary address via either a DIP switch or via front panel selectors. This primary address is the same one used in the VISA resource string to `viOpen()`. The simplest and most common GPIB resource string is "GPIB::<primary address>::INSTR". Recall that the "INSTR" resource class informs VISA that you are doing instrument (device) communication. Most GPIB programs perform simple message-based transfers (write command, read response). For more information about VISA message-based functionality, see Message-Based Communication.

There are several VISA attributes specific to the GPIB INSTR resource. The `VI_ATTR_GPIB_PRIMARY_ADDR` and `VI_ATTR_GPIB_SECONDARY_ADDR` attributes are read-only, and these return the same values that were used in the resource string passed to `viOpen()`. If the specified device does not have a secondary address, that attribute query will succeed and return a value of -1. The attribute `VI_ATTR_GPIB_READADDR_EN` controls whether each message to or from the same device will cause the driver to readdress the device. This attribute is true (enabled) by default, and disabling this attribute (setting it to false) may provide a slight performance increase by removing unnecessary bus-level readdressing to the same device. The attribute `VI_ATTR_GPIB_UNADDR_EN` controls whether the driver will follow each message to or from the specified device with untalk (UNT) and unlisten (UNL) commands. This attribute is false (disabled) by default, which is the most optimal setting. Changing the values of these attributes may be necessary for certain older non-IEEE 488.2-compliant devices.

More complex GPIB systems often include multiple GPIB controllers (or boards) and devices with both primary and secondary addresses. The canonical form of a complex GPIB instrument resource string is "GPIB<controller>::<primary address>::<secondary address>::INSTR". The controller number is the same as used in the GPIB configuration utility (MAX on Windows, the GPIB Control Panel applet on Macintosh, or `ibconf` on UNIX). If not specified, the controller



number defaults to 0.

Since the NI-VISA and NI-488.2 APIs are very similar and both provide the same GPIB functionality, which should you choose? If you are already familiar with NI-488.2 and are programming only GPIB devices, then there is not a strong reason for you to change to VISA. NI-488.2 is supported in all major application development environments, including LabVIEW and Measurement Studio. However, if you have instruments with more than one type of port or connection available to them, then using VISA might be advantageous because you can use the same API regardless of the connection medium.

Finally, many modern instrument drivers rely on VISA for their I/O needs, so if you are using instrument drivers, then you need to at least install NI-VISA for them to be able to execute.





### Related concepts:













- [VISA Message-Based Communication](#)

## Comparison Between NI-VISA and NI-488.2 APIs

For GPIB users who are familiar with NI-488.2, the following table shows several common, but not all, NI-488.2 device-level function calls and the corresponding VISA operations. As you can see, the APIs are almost identical. The difference is that VISA is extensible to additional hardware interfaces. Therefore, if you are programming multiple devices that communicate over more than one bus type, it might be easier to use VISA for your entire system.

Table 2. NI-VISA and NI-488.2 Functions and Operations

| C NI-488.2 Device Function | C VISA INSTR Operation | LabVIEW NI-488.2 Device Function   | LabVIEW VISA INSTR Operation   |
|----------------------------|------------------------|--|--|
| ibdev                      | viOpen                 | <no equivalent>  |  VISA Open  |
| ibonl                      | viClose                | <no equivalent>  |  VISA Close |
| ibwrt                      | viWrite                |  GPIB Write |  VISA Write |

| C NI-488.2 Device Function | C VISA INSTR Operation | LabVIEW NI-488.2 Device Function  | LabVIEW VISA INSTR Operation  |
|----------------------------|------------------------|---|---|
| ibrd                       | viRead                 |  GPIB Read           |  VISA Read           |
| ibclr                      | viClear                |  GPIB Clear          |  VISA Clear          |
| ibtrg                      | viAssertTrigger        |  GPIB Trigger        |  VISA Assert Trigger |
| ibrsp                      | viReadSTB              |  GPIB Serial Poll    |  VISA Read STB       |
| ibwait                     | viWaitOnEvent          |  Wait for GPIB RQS   |  Wait for RQS        |
| ibconfig                   | viSetAttribute         |  GPIB Initialization |  VISA Property Node  |

One difference in the event mechanism between NI-488.2 and VISA is worth noting. In VISA, you must always call `viEnableEvent()` prior to being allowed to receive events. While this was not the case with NI-488.2, this is required in VISA to avoid the race condition of trying to wait on events for which the hardware may not be enabled. Thus, you should enable the session for events not just immediately before calling `viWaitOnEvent()`, but before the device has even been triggered or configured to generate a service request event.

## Board-Level Programming

Advanced users occasionally need to control multiple devices simultaneously or need to have multiple controllers connected together in a single system. Power GPIB programmers use interface-level (bus-level) commands to do this. The corresponding VISA resource for this is the GPIB INTFC resource, and the form of the resource string is "GPIB<controller>::INTFC". This allows raw message transfers in which the driver does not perform automatic device addressing, as it does with INSTR. Also, with the INTFC resource, the controller can directly query and manipulate specific lines on the bus such as SRQ or NDAC, and also pass control to other devices that have controller capability.

For users who are familiar with NI-488.2, the following table shows several common, but not all, NI-488.2 board-level function calls and the corresponding VISA operations.

Table 3. Board-Level Programming Functions and Operations

| NI-488.2 Board Function | VISA INTFC Operation |
|-------------------------|----------------------|
| ibfind                  | viOpen               |
| ibonl                   | viClose              |
| ibwrt                   | viWrite              |
| ibrd                    | viRead               |
| ibwait                  | viWaitOnEvent        |
| ibconfig                | viSetAttribute       |
| ibask, ibwait           | viGetAttribute       |
| ibcmd                   | viGpibCommand        |
| ibsre                   | viGpibControlREN     |
| ibgts, ibcac            | viGpibControlATN     |
| ibsic                   | viGpibSendIFC        |

For users who need to write an application that will run inside a device, such as firmware, the INTFC resource provides the necessary functionality. The device status byte attribute is useful for reflecting application status.

# Programming VXI Devices in VISA

A VXI device has a unique logical address, which serves as a means of referencing the device in the VXI system. This logical address is analogous to a GPIB primary address. VISA uses an 8-bit logical address, allowing for up to 256 VXI devices in a VXI system. VISA addresses a specific VXI device with a resource string identifying the VXI system that the device is in and the logical address of this particular device:  
`"VXI<system>::<logical address>::INSTR"`.

Each VXI device has a specific set of registers, called configuration registers. See the NI-VXI online help for a diagram. These registers are located in the upper 16KB of the 64KB A16 address space. The logical address of a VXI device determines the location of the device's configuration registers in the 16KB area reserved by VXI. The rest of A16 space is available for VME devices. The 16MB A24 address space and the 4GB A32 address space are available for VXI and VME devices. Each VXI system has a Resource Manager which is responsible for allocating each device's requests in the appropriate address space. When you open a VXI/VME INSTR resource in VISA, you have access to registers in the spaces that have been allocated by the Resource Manager for the device corresponding to that INSTR resource. Devices which provide only this minimal level of capability are called register-based devices, and support VISA operations such as `viInX/viOutX` (read/write a single register), `viMoveInX/viMoveOutX` (perform a block move to read or write a block of registers), `viMapAddress` (map a region of VXI memory into your application for low-level access), and others.

In addition to register-based devices, the VXIbus specification also defines message-based devices, which are required to have communication registers in addition to configuration registers. All message-based VXIbus devices, regardless of the manufacturer, can communicate using the VXI-specified Word Serial Protocol. In addition, you can establish higher-performance communication channels, such as the shared-memory channels in Fast Data Channel (FDC), to take advantage of the VXIbus bandwidth capabilities (a diagram of these protocols is shown in the NI-VXI online help).

The VXIbus Word Serial Protocol is a standardized message-passing protocol. This protocol is functionally very similar to the IEEE 488 protocol, which transfers data messages to and from devices one byte at a time. Thus, VISA message-based devices

communicate in a fashion very similar to GPIB instruments. In general, message-based devices typically contain a higher level of local intelligence that uses or requires a higher level of communication. In addition, the Word Serial Protocol has special messages for configuring message-based devices. All VXI message-based devices are required to support the Word Serial Protocol and support a basic level of standard communication. There are even higher level message-based protocols, such as Standard Commands for Programmable Instrumentation (SCPI); these are not required protocols, and not all VXI message-based devices support them. Message-based VXI devices support VISA operations such as `viRead/viWrite` (Word Serial read/write buffer), `viClear` (Word Serial clear), `viPrintf/viScanf` (formatted I/O), `viAssertTrigger` (Word Serial trigger), `viVxiCommandQuery` (Word Serial command and/or response), and others.

Since the VISA API is very similar to the NI-VXI API, and both provide almost the same VXI functionality, which should you choose? NI recommends using the VISA API because it allows you to control multiple VXI systems (controllers) from a single computer, provides a more flexible API that allows you to move to other interfaces if the application demands it, and usually provides equal or better performance. However, if your application already uses NI-VXI and you are programming only VXI devices, then there is not a strong reason for you to change the application to VISA. For new applications, though, VISA is almost always preferred. Finally, most modern instrument drivers rely on VISA for their I/O needs, so if you are using instrument drivers, then you need to at least install NI-VISA for them to be able to execute.

### Related concepts:

- [VISA Register-Based Communication](#)
- [VISA Message-Based Communication](#)

## VXI/VME Interrupts and Asynchronous Events in VISA

VXI/VME devices can communicate asynchronous status and events through VXI/VME interrupt events (`VI_EVENT_VXI_VME_INTR`) or by using specific messages called signals (`VI_EVENT_VXI_SIGP`). Since VXI interrupts can be treated just like signals, a VISA application for VXI devices will typically just use `VI_EVENT_VXI_SIGP` to handle both interrupts and signals, regardless of which is actually sent in hardware. The main difference is that the status/ID returned as an attribute of the event is 16-bit

for `VI_EVENT_VXI_SIGP` and 32-bit for `VI_EVENT_VXI_VME_INTR`.

The VXI specification also makes use of triggering (`VI_EVENT_TRIG`) to synchronize events between VXI devices. VXI devices support these events in the INSTR resource through the standard VISA operations such as `viEnableEvent`, as discussed in VISA Events. Since devices can both send and receive triggers, the attribute `VI_ATTR_TRIG_ID` specifies the line used for either. You cannot use the same session to both assert and receive triggers; for this, you need multiple sessions.

### Related concepts:

- [VISA Events](#)

## Performing Arbitrary Access to VXI Memory with VISA

VISA provides the VXI MEMACC resource class to allow access to arbitrary locations in VXI address spaces. When you open a VXI INSTR resource, VISA automatically performs all register I/O in the address spaces used by that device relative to that device's memory region, and will prevent accidental access outside of the region allocated for your device. If you need to access a memory region not associated with a particular device, or use a low-level scheme for performing your register I/O that uses absolute addresses, you should use the MEMACC resource which provides this capability. When using a MEMACC resource, all address parameters are absolute within the given address space; knowing a device's base address is both required by and relevant to the user. The VISA resource string format for this is `"VXI<system>::MEMACC"`. You can still use the same VISA operations for performing register I/O enumerated above, such as `viInX/viOutX`, `viMoveInX/viMoveOutX`, and `viMapAddress`.

## Other VXI Resource Classes and VISA

For certain applications, such as asserting interrupts or triggers, it may be necessary to access the VXI mainframe or chassis ("backplane") directly. VISA provides the BACKPLANE resource for this purpose, where each VXI mainframe can be accessed using the VISA resource string `"VXI<system>::<mainframe number>::BACKPLANE"`. The BACKPLANE resource encapsulates the operations and properties of each mainframe (or chassis) in the VXI system, and lets a controller query and manipulate specific lines on a specific mainframe in a given VXI system. The operations `viMapTrigger`,






























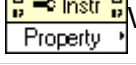
`viUnmapTrigger`, `viAssertTrigger`, and the event `VI_EVENT_TRIG` supported on this resource allow the user to map, unmap, assert, and receive hardware triggers. You can also use `viAssertUtilSignal`, `viAssertIntrSignal`, `VI_EVENT_VXI_VME_SYSFAIL`, and `VI_EVENT_VXI_VME_SYSRESET` to assert and receive various utility and interrupt signals. This includes advanced functionality that might not be available in all implementations or on all controllers.

It is possible to configure your VXI controller to be a Word Serial servant in your VXI system, with another controller as its commander. For such situations, VISA provides another class of asynchronous events associated with the Word Serial protocol: the Word Serial Servant protocol. Using the VISA SERVANT resource, your device can act as a servant, which means that it can use `VI_EVENT_IO_COMPLETION` to respond to requests from a Word Serial commander. This resource is accessed using `"VXI<system>::SERVANT"` and encapsulates the operations and properties of the capabilities of a device and a device's view of the system in which it exists. The SERVANT resource exposes the device-side functionality of the device associated with the given resource. This functionality is somewhat unusual for a VXI controller, and in most cases you will never need to use the SERVANT resource. The SERVANT resource provides the complementary functions for the message-based operations discussed above, and therefore implements the servant side `viRead`, `viWrite`, etc. for buffer reads and writes, `viPrintf`, `viScanf`, etc. for formatted I/O, and asynchronous message-based notification events. The resource also provides the ability to assert and receive interrupt and utility signals.

## Comparison Between NI-VISA and NI-VXI APIs

As a VXI programmer you may be familiar with the NI-VXI API, but NI recommends that all new VXI applications be developed in NI-VISA, which provides additional flexibility, features, and performance. Fortunately, translating NI-VXI API code to VISA is made fairly simple by the close correlation between the two APIs. For users who are familiar with the NI-VXI API, the following table shows several common, but not all, NI-VXI API function calls and the corresponding VISA operations. You can see that the APIs are almost identical. The difference is that VISA is extensible to additional hardware interfaces. Therefore, if you are programming multiple devices that communicate over more than one bus type, it might be easier to use VISA for your entire system.

Table 4. NI-VISA and NI-VXI Functions and Operations

| C NI-VXI Function    | C VISA INSTR Operation     | LabVIEW NI-VXI Function  | LabVIEW VISA INSTR Operation  |
|----------------------|----------------------------|--|---|
| InitVXIlibrary       | viOpenDefaultRM,<br>viOpen |  InitVXIlibrary   |  VISA Open   |
| CloseVXIlibrary      | viClose                    |  CloseVXIlibrary  |  VISA Close  |
| WSwrt                | viWrite                    |  WSwrt  |  VISA Write  |
| WSrd                 | viRead                     |  WSrd   |  VISA Read   |
| WSclr                | viClear                    |  WSclr  |  VISA Clear  |
| WStrg, SrcTrig       | viAssertTrigger            |  WStrg,  SrcTrig |  VISA Assert Trigger   |
| VXlin, VXlout        | viInX, OutX                |  VXlin,  VXlout  |  VISA InX,  VISA OutX               |
| VXImove              | viMoveInX,<br>viMoveOutX   |  VXImove  |  VISA Move InX,  VISA Move OutX |
| MapVXIAddress        | viMapAddress               |  MapVXIAddress  |  VISA Map Address  |
| AssertVXIint         | viAssertIntrSignal         |  AssertVXIint   |  VISA Assert Interrupt   |
| EnableVXItoSignalInt | viEnableEvent              |  EnableVXItoSignalInt   |  VISA Enable Event   |
| WaitForSignal        | viWaitOnEvent              |  WaitForSignal  |  VISA Wait on Event  |
| GetDevInfo           | viGetAttribute             |  GetDevInfoLong   |  VISA Property Node  |

An important difference between the NI-VXI API and VISA is the scope of the effect of certain function calls. In the NI-VXI API, many functions (notably, enabling for events) acted on the VXI controller directly and therefore applied to the entire VXI system. Since VISA is generally device-oriented rather than controller-oriented, the corresponding VISA INSTR operations act on a specific VXI device, not the entire



system.

# Programming PXI Devices in NI-VISA

NI-VISA supports programming PCI and PXI (PCI eXtensions for Instrumentation) devices plugged into the local PC or PXI chassis, or PXI devices in a remote chassis connected via a remote controller such as MXI-3

Users who are writing an application for a PCI or PXI card can use NI-VISA to gain full access to all the device's configuration, I/O, and memory-mapped registers. NI-VISA currently supports the PXI interface on Windows, LabVIEW RT (Phar Lap ETS), and Linux. The supported functionality is identical for PCI and PXI cards. The terms PCI and PXI are used somewhat interchangeably in this section; technically, PXI is a rigorously defined extension of PCI.

To use PXI or PCI devices in your program, make sure you define the macro `"NIVISA_PXI"` before including `"visa.h"`.

A PXI resource is uniquely identified in the system by three characteristics: the PCI bus number on which it is located, the PCI device number it is assigned, and the 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 will be in the range 0–7. The device number is associated with the slot number, but these numbers are usually different. 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's CPU. The canonical resource string that you pass to `viOpen()` for a PCI or PXI device is `"PXI<bus>::<device>::<function>::INSTR"`, but based on the previous explanation, this can be difficult to determine.

A better way to determine the resource string is to query the system with `viFindRsrc()` and use or display the resource(s) returned from that operation. Each PCI device has a vendor code and a model code; this is much the same as VXI does, although the vendor IDs are different. You can create a query to search for devices of a particular attribute value; in this case, you can search for a specific vendor ID and model code. For example, the PCI vendor ID for National Instruments is 0x1093. If NI made a device with the model code 0xBEEF, you could call `viFindRsrc()` with the expression `"PXI?*INSTR{VI_ATTR_MANF_ID==0x1093 && VI_ATTR_MODEL_CODE==0xBEEF}"`. In many cases, the returned list has one or

only a few devices.

NI-VISA provides a convenient means of accessing advanced functionality of PCI and PXI devices. The alternative to using NI-VISA for PCI or PXI device communication is writing a kernel driver. By using NI-VISA, you avoid having to learn how to write kernel drivers, you avoid having to learn a different kernel model for each Windows operating system, and you gain platform independence and portability by scaling to other operating systems such as LabVIEW RT now and others in the future.

## User-Level Functionality

An INSTR session to a PCI or PXI device provides the same register-level programming functionality as in VXI. NI-VISA supports both high-level and low-level 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 support any or all of the BARs. This information is device dependent but can be queried through the attributes VI\_ATTR\_PXI\_MEM\_TYPE\_BAR0–VI\_ATTR\_PXI\_MEM\_TYPE\_BAR5. The values for this attribute 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 supported region.

In addition to register accesses, NI-VISA supports the event VI\_EVENT\_PXI\_INTR to provide notification to an application that the specified device has generated a PCI interrupt. This event allows a user to write an entire device driver or instrument driver at the user level, without having to write any kernel code.

### Related concepts:

- [VISA Register-Based Communication](#)

## Configuring NI-VISA to Recognize a PXI Device

Each PCI device must have a kernel level driver associated with it; this is done in Windows via a `.inf` file. For NI-VISA to recognize your device, you must run the NI-VISA Driver Wizard, available via the **Start » National Instruments » VISA** menu.

The wizard first prompts you for basic information NI-VISA needs to properly locate your PXI instrument. This includes the following:

- **Instrument Prefix**—The VXIplug&play or IVDI instrument driver prefix for the device.
- **PXI Manufacturer ID**—This 16-bit value is vendor specific and is unique among PCI-based device providers. The vendor ID number for National Instruments, for example, is 0x1093. If the product vendor uses a commercial PCI core, this value would be the vendor ID of the PCI core component.
- **PXI Model Code**—The 16-bit device ID value is device specific, defined by the instrument provider, and required for PCI-based devices. If the product vendor uses a commercial PCI core, this value would be the device ID of the PCI core component.
- **Subsystem Manufacturer ID**—This 16-bit value is vendor specific and is unique among PCI-based device providers. If this value exists, it specifies the vendor ID of the actual product. This value may be the same as the primary PXI Manufacturer ID.
- **Subsystem Model Code**—The 16-bit device ID value is device specific, defined by the instrument provider, and required for PCI-based devices.
- **Generates interrupts**—Checking this box indicates that you want to use the VISA event-handling model in response to hardware interrupts your PXI instrument generates.

If the device vendor has provided you with a Module Description File (also called a `module.ini` file), you can import the information from that file into the wizard instead of entering these settings yourself. Module Description Files provide a mechanism for informing the operating system and the PXI Resource Manager about key attributes of a PXI module. Refer to the PXI Systems Alliance website for the specification for these files.

In text boxes where numerical information is required, preceding the number with `0x` designates a hexadecimal value. The wizard assumes all other numeric entries are decimal values.

If you need to handle hardware interrupts, check **Generates interrupts** and the wizard guides you through a two-step process. In Step 1, you specify how your device detects a pending interrupt. This is done via one or more register accesses, where each access is a single register read or write of a specified width to a given offset relative to a given address space. In the wizard, you specify each access as a Read, Write, or Compare.

The Compare operation is essential for determining whether a PCI/PXI device is interrupting. A Compare operation performs a Read, then applies a user-specified mask to the result and compares the masked result with another user-specified value (you specify both of these values in the wizard). In order to determine whether your device is interrupting, the Compare operation has an associated result of True or False. NI-VISA decides that the device is interrupting if and only if the result of all Compare operations is True. Because NI-VISA relies on the result of the Compare operation in making this determination, at least one Compare operation must be present in an interrupt detection sequence for the sequence to be valid.

If your device has multiple potential interrupt sources, you can specify multiple interrupt detection sequences. At least one sequence must be considered valid for NI-VISA to deem that your device is interrupting.

In addition to the interrupt detection sequence, NI-VISA also needs the sequence of register operations required to acknowledge an interrupt condition for your device; this is Step 2. At interrupt time, if NI-VISA determines that your device is interrupting (as discussed above), this second sequence should do whatever is necessary to squelch the interrupt condition. This sequence is constructed using the same Read, Write, and Compare operations discussed in Step 1, and individual operations are entered in an identical manner. Because this sequence should consist of the minimum operations necessary to turn off an interrupt condition for your device, the result of any Compare operations, while still valid, are irrelevant to interrupt acknowledgment. If your device uses ROAK (Release on Interrupt Acknowledge) interrupts, and the ROAK register was accessed in the sequence specified by Step 1, this sequence can be left blank.

The wizard will also allow you to enter certain Windows Device Manager settings; these are cosmetic and do not affect the ability of NI-VISA to recognize and control your PXI instrument. They are provided as a convenience, allowing you to more fully customize your instrument driver package.

When you are done, the NI-VISA Driver Wizard generates a Windows Setup Information (`.inf`) file for each supported operating system. Before a PXI device will be visible to NI-VISA, you must use the `.inf` files to update the Windows system registry. The procedure for using a `.inf` file to update the registry is Windows-version dependent. To manually install a `.inf` file on any machine, including the one on which it was generated, open the appropriate `.inf` file in a text editor and follow the instructions

on the first few lines at the top. Alternately, you can let the wizard install the `.inf` file appropriate for your machine before the wizard exits.

## Using LabWindows/CVI to Install Your Device `.inf` Files

LabWindows/CVI supports distribution of `.inf` files for PXI/PCI and USB devices. For VISA-based instrument drivers for devices of these types, you must include the generated `.inf` file in your application's installer. In LabWindows/CVI 8.1.1 and later, perform the following steps to create the distribution:

1. Generate the `.inf` file for your instrument using the NI-VISA Driver Wizard.
2. In the LabWindows/CVI Edit Installer dialog box Files tab, create an installation directory for the `.inf` file. NI recommends you choose a specific and meaningful directory path, such as `[Windows Volume]\<Company Name>\Drivers\<Driver Name>\<Revision>` or `[VXI PnP OS]\<Driver Name>\<Revision>`. Your `.inf` file must not be installed to a directory containing any other `.inf` files, or installation may fail. You should not explicitly install your file to the Windows `.inf` store; it will be implicitly registered and copied to the Windows `.inf` store during installation. If you have multiple `.inf` files, create a separate directory for each `.inf` file.
3. Add your `.inf` file to the directory you just created.
4. Build the distribution.



**Note** If you are creating `.inf` files for a USB device and want to support both Windows XP/Server 2003 R2 and Windows 10/8.1/7 SP1/Vista 32/Vista 64/Server 2008 R2 editions, you must create two separate LabWindows/CVI installers, one for each `.inf` file generated.

## Other PXI Resource Classes and VISA

For certain applications, such as mapping triggers, it may be necessary to access the PXI chassis ("backplane") directly. VISA provides the BACKPLANE resource for this purpose, where each PXI chassis is accessed using the VISA resource string `"PXI<system>::<chassis number>::BACKPLANE"`. The BACKPLANE resource encapsulates the operations and properties of each chassis in the PXI system, and lets a controller query and manipulate specific attributes of and lines on a specific

chassis in a given PXI system. The operations `viMapTrigger`, `viUnmapTrigger`, and `viAssertTrigger` supported on this resource allow the user to map, unmap, reserve, or unreserve hardware trigger resources. The controller can also query attributes such as the manufacturer and model of a chassis. This includes advanced functionality that might not be available in all implementations or on all controllers.

## Using the NI-VISA Driver Wizard and NI-VISA to Register-Level Program a PXI/PCI Device under Windows

You can use NI-VISA to program PCI and PXI devices installed in a PC or PXI chassis. By writing an application for a PCI or PXI device with NI-VISA, you 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.

This tutorial explains how to use NI-VISA and the NI-VISA Driver Wizard to develop a low-level driver for a PXI/PCI device. It describes the NI-VISA features you can use to register-level program PXI/PCI devices. To demonstrate how to use the VISA API for this purpose, the tutorial includes examples using a National Instruments E Series PXI data acquisition module, the NI PXI-6070E. This module is included as a tool to demonstrate NI-VISA features; therefore, there is no additional register-level information about this module. 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 tutorial introduces the NI-VISA event-handling model for handling interrupts from a PXI/PCI device. It also explains how to use LabWindows/CVI to install the Windows setup files you create for your device and describes the NI-VISA API PXI functionality.

## PXI 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 the highest performance measurement and automation platform available today.

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 device register set, 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. The NI-VISA Driver Wizard, 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 NI-VISA Driver Wizard replaced the PXI Driver Development Wizard.

## 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 a `.inf` file. To access the NI-VISA Driver Wizard, select **Start » National Instruments » VISA**.

This section explains how to configure NI-VISA to recognize a PXI/PCI device.



## 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 shown in the following figure.

Because you want to control a PXI or a PCI device, select **PXI/PCI** and click **Next**.

## Basic Device Information

The NI-VISA Driver Wizard first prompts for basic information NI-VISA needs to properly locate and identify a PXI/PCI device. This basic information, such as manufacturer identification and model code, should be documented in the register-level programming information for your PXI/PCI device. The following figure shows the Basic Device Information window of the NI-VISA Driver Wizard.

Figure 1. Basic Device Information Window

**NI-VISA Driver Wizard**

**PXI/PCI - Device Information**

Welcome to the VISA Driver Development Wizard for PXI/PCI! This wizard gathers the necessary information to allow NI-VISA to control your PXI or PCI device. Please enter the requested information about your device.

This wizard will generate an INF file for use with Windows 2000/XP, LabVIEW RT, Mac OS X, and Linux. The INF file tells the operating system to allow NI-VISA to control the PXI or PCI device that you specify here.

|                       |                           |                      |
|-----------------------|---------------------------|----------------------|
| Manufacturer ID (VID) | Subsystem Manufacturer ID | Manufacturer Name    |
| 0x 1093               | 0x 0000                   | National Instruments |
| Model Code (PID)      | Subsystem Model Code      | Model Name           |
| 0x 11B0               | 0x 0000                   | PXI-6070E            |

☐ This device uses a subsystem  
☒ This device generates interrupts  
☐ This device uses PXI Express

Load Settings from a Module Description File...

< Back    Next >    Cancel    Help

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

- **Manufacturer ID**—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**—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 is used in this tutorial, is 0x11B0.
- **This device generates interrupts**—Some PXI/PCI devices generate interrupts to request attention. Checking this box indicates that you must use the VISA event-handling model in response to hardware interrupts your PXI/PCI device generates. For the purposes of demonstrating the NI-VISA Driver Wizard, the **This device generates interrupts** box is checked, although there are no examples of using interrupts on the PXI-6070E, because this is beyond the scope of this tutorial.
- **This device uses PXI Express**—PXI Express devices provide to software a way to read the slot number. By checking this box, you can specify the sequence of register accesses necessary to read the slot number from a PXI Express device, or a PXI device that supports this feature.
- **This device uses a subsystem**—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 item is checked by default, because the PCI specification now requires devices to support the subsystem registers.
- **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 must 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.

In addition to manually entering all basic device and the interrupt information, the NI-VISA Driver Wizard includes 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.

To obtain the values for these settings, contact your PXI/PCI device manufacturer. Along with the `.inf` files, the NI-VISA Driver Wizard also creates a `module.ini` file you can distribute with your PXI instrument. When all information is loaded, click **Next**.

The next window depends on whether you checked **Generates interrupts** or **PXI Express**. If your device does not generate interrupts, you can skip to Output Files Generation.

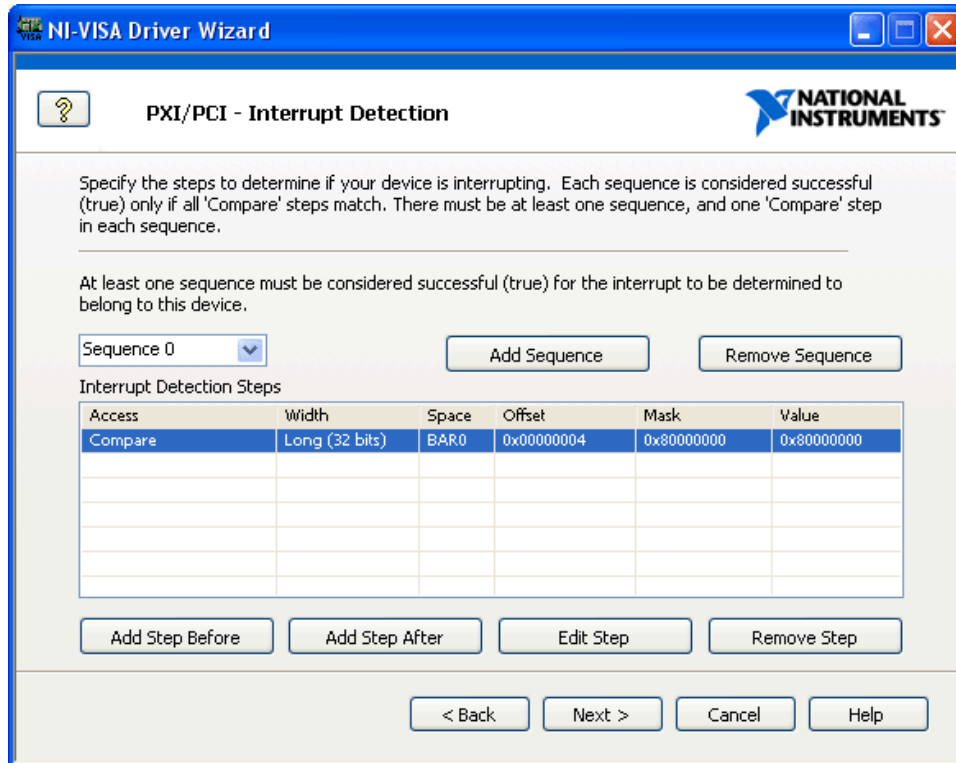
#### Related concepts:

- [Output Files Generation](#)

## Interrupt Detection Information

If the **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 Information 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. The following figure shows the Interrupt Detection Information window.

Figure 2. Interrupt Detection Information Window



You can use a Read/Compare operation to determine whether your device is asserting a hardware interrupt. 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 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. The following figure 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 sequence has 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 device register documentation. You can add and remove sequences using the **Add Sequence** and **Remove Sequence** buttons, respectively.

Figure 3. Interrupt Information Window

|   |                     |
|---|---------------------|
| Access Type   | Access Width        |
| Compare   | Long (32 bits)      |
| Address Space   | Space Offset        |
| BAR0  | 0x 00000014         |
| Compare Mask  | Write/Compare Value |
| 0x 80000000   | 0x 80000000         |
| <input type="button" value="OK"/> <input type="button" value="Cancel"/> <input type="button" value="Help"/> |                     |

The above example specifies that to make this determination, you must read a 32-bit value from BAR0 at offset 0x14. NI-VISA must then check this value to determine the status of bit 31 (highest order bit in the register). If bit 31 is high, NI-VISA knows the device is generating an interrupt. Using the Compare mask, you can mask in the particular bits you need to compare, in this case bit 31. The hexadecimal value that corresponds to bit 31 being high is 0x80000000. The Value to write or compare is the value that you expect the 32-bit register to be equal to after applying the mask. If you were doing a Write instead of a Read/Compare, you 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 **Next** to continue with the wizard.

## Interrupt Removal Information

If the **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 4. Interrupt Removal Information

Specify the steps to acknowledge an interrupt from your device, so as to remove the interrupt condition from the PCI bus. If the device is ROAK (release on acknowledge) and the register was already read, no steps may be required.

| Access | Width          | Space | Offset     | Mask | Value  |
|--------|----------------|-------|------------|------|--------|
| Write  | Word (16 bits) | BAR0  | 0x00000010 |      | 0xDE01 |
|        |                |       |            |      |        |
|        |                |       |            |      |        |
|        |                |       |            |      |        |
|        |                |       |            |      |        |

Buttons: Add Step Before, Add Step After, Edit Step, Remove Step, < Back, Next >, Cancel, Help

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 device register documentation.

After selecting one of the **Add a step** buttons, the PXI Interrupt Information window appears. The following figure specifies that to remove the interrupt, you must write a 16-bit value to BAR0 at offset 0x10. The Value to write or compare is the value you will write to the 16-bit register. Again, this information is provided for example purposes only and is not useful when attempting to handle interrupts for the PXI-6070E. Click **OK** when you finish entering a particular interrupt detection step.

Figure 5. Interrupt Information Window

|               |                     |
|---------------|---------------------|
| Access Type   | Access Width        |
| Write         | Word (16 bits)      |
| Address Space | Space Offset        |
| BAR0          | 0x 00000010         |
| Compare Mask  | Write/Compare Value |
| 0x 0000       | 0x DE01             |

OK Cancel Help

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

## Interrupt Disarm Information

If the **Generates interrupts** box was checked, the next window is the Interrupt Disarm window. NI-VISA allows you to specify a sequence of register operations to disarm interrupts on your device if a process terminates abnormally. (When a process terminates abnormally, it does not disarm interrupts. This leaves the system vulnerable to receiving an errant interrupt from a device that no longer has an interrupt handler, which could cause a blue screen or system hang.) NI-VISA executes the specified sequence of register operations only if the crashing process is the last process using the device.

Figure 6. Interrupt Disarm Information Window

**NI-VISA Driver Wizard**

**PXI/PCI - Interrupt Disarm**

Specify the steps to stop your device from interrupting, so that NI-VISA can perform these steps in the event of abnormal process termination. Upon normal process termination, the user is responsible for disarming interrupts.

Interrupt Disarm Steps

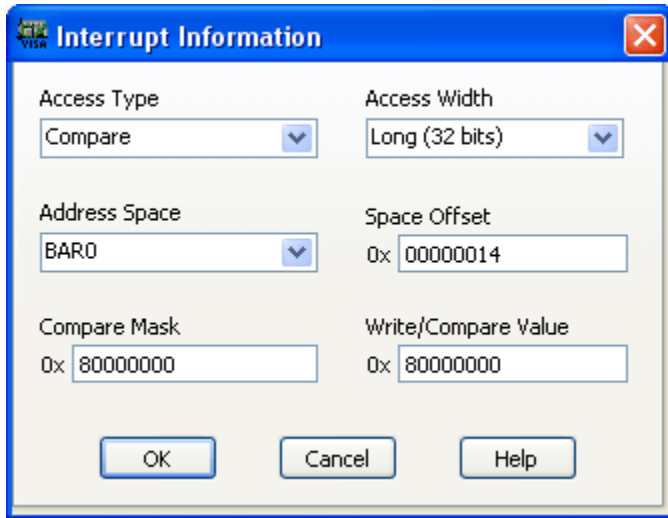
| Access  | Width          | Space | Offset     | Mask       | Value      |
|---------|----------------|-------|------------|------------|------------|
| Compare | Long (32 bits) | BAR0  | 0x00000014 | 0x80000000 | 0x80000000 |
|         |                |       |            |            |            |
|         |                |       |            |            |            |
|         |                |       |            |            |            |
|         |                |       |            |            |            |

Just as recognizing and acknowledging an interrupt may take multiple steps, disarming a device may also take multiple steps. To add a step, click the **Add a Step Before** or **Add a Step After** button. The steps to disarm your PXI/PCI device from interrupting should be in the device register documentation.

After selecting one of the **Add a Step** buttons, the Interrupt Information window appears.



Figure 7. Interrupt Information Window

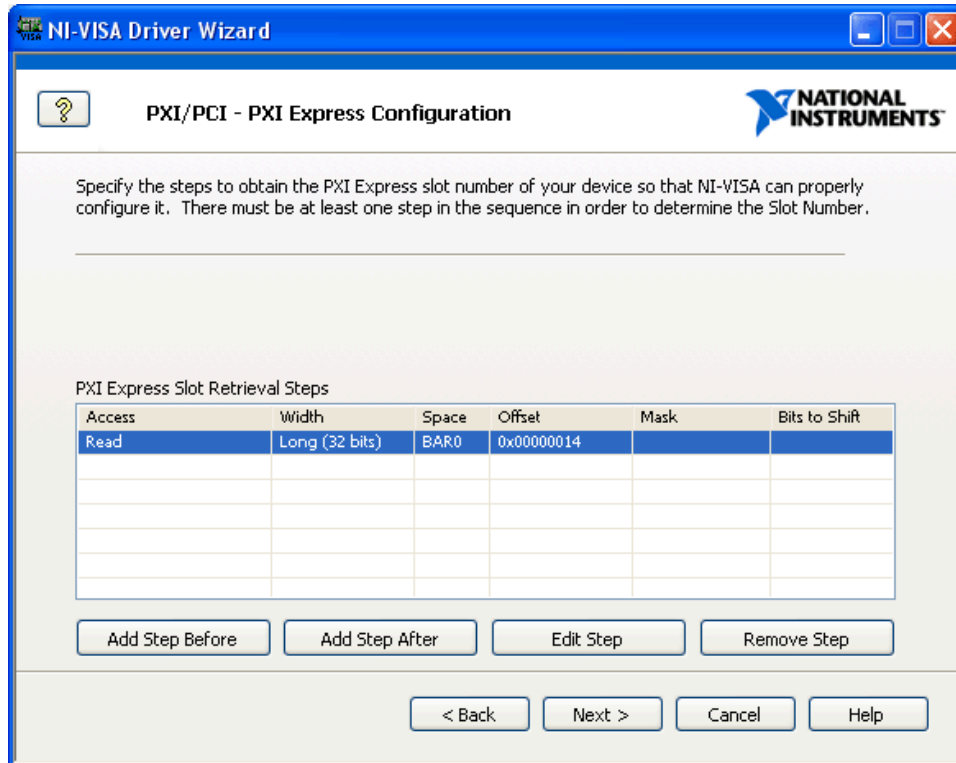


There may be more than one step documented for your device. In this case, continue to add steps until the sequence for your device is complete. When you have added all steps for your device, click **Next** to continue with the wizard.

## PXI Express Configuration Information

If the **Supports PXI Express features** box is checked, the next window is the PXI Express Configuration window. PXI Express requires devices to be able to report their slot number. Using this window, you can specify the steps required to query this value from the device.

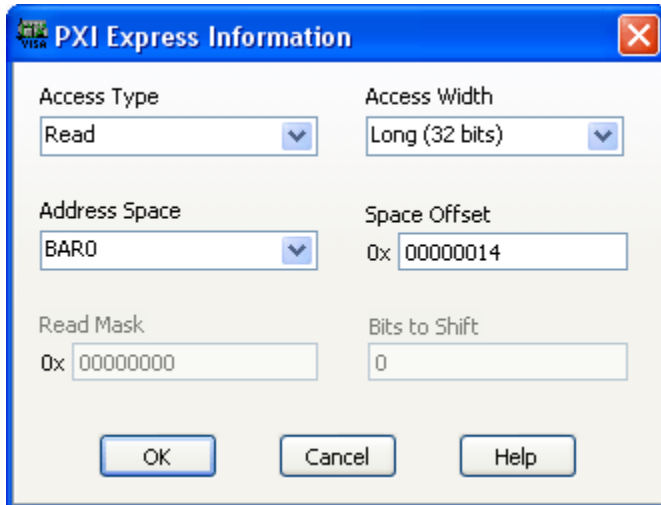
Figure 8. PXI Express Configuration Information Window



Typically, requesting the slot number from a device involves reading a register. This may involve multiple steps, as a single register may have multiple purposes depending on the last value written to it, or the state of other registers. To add a step, click the **Add a step before** or **Add a step after** button. The steps to query the slot number from your device that supports PXI Express features should be in the device register documentation.

After selecting one of the **Add a step** buttons, the PXI Express Information window appears. The following figure specifies that to get the slot number, you must read a 32-bit value from BAR0 at offset 0x14. The **Read Mask** and **Bits to Shift** values are used, frequently in combination, to select specific bits that represent the slot number when the value returned by a register contains more information than just a slot number. Again, this information is provided for example purposes only and is not useful when attempting to handle interrupts for the PXI-6070E. Click **OK** when you finish entering a particular register access step.

Figure 9. PXI Express Information Window

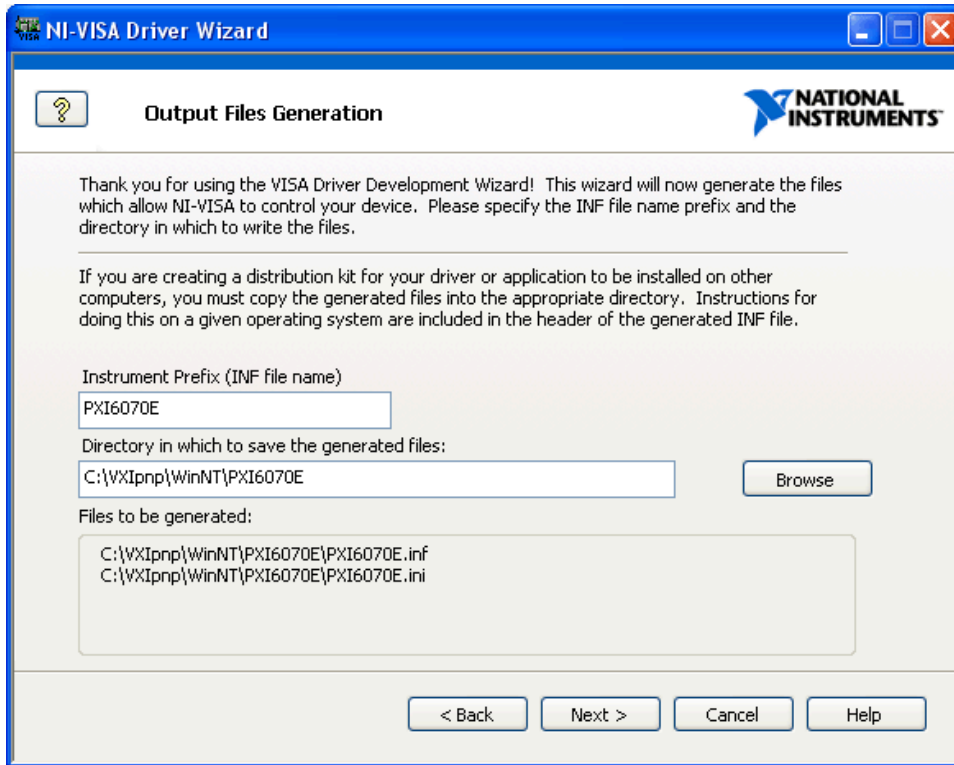


There may be more than one step documented for your device. In this case, continue to add steps until the sequence for your device is complete. When you have added all steps for your device, click **Next** to continue with the wizard.

## Output Files Generation

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

Figure 10. Output Files Generation



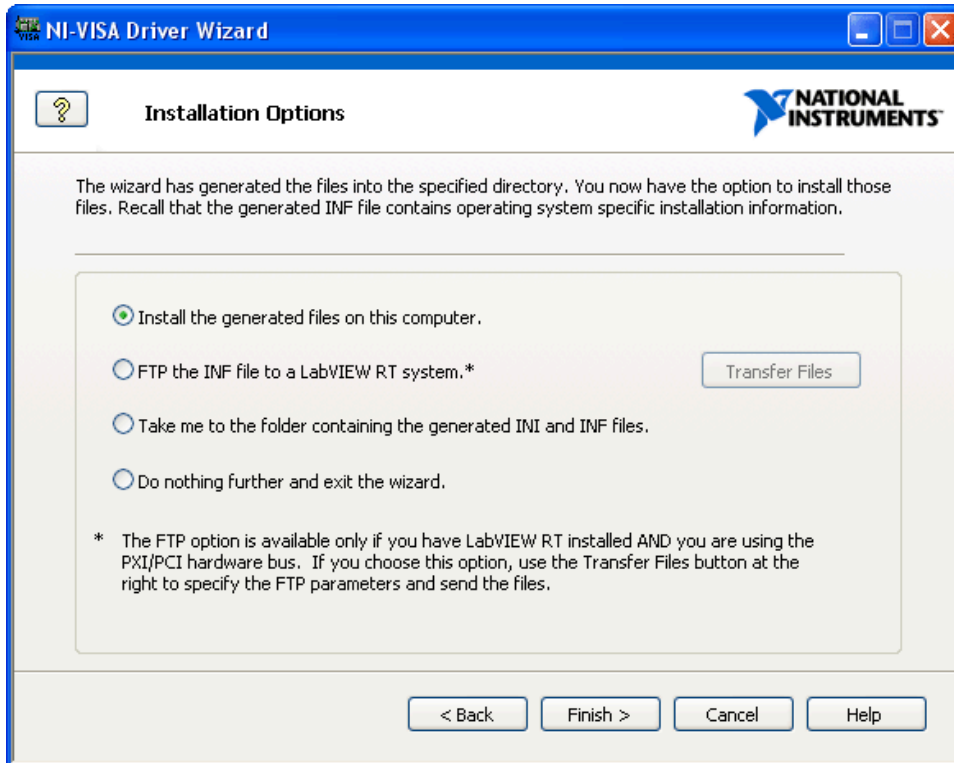
The **PXI Instrument Prefix** is a descriptor you use to identify the files used for this device. Enter a PXI instrument prefix, select the desired directory in which to place these files, and click **Next**. The `.inf` file is created in the directory specified by the output file directory, and the Installation Options window appears as shown in next topic.

## Installation Options

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

The NI-VISA Driver Wizard can automatically install the `.inf` file for your device. To automatically install the required files, select **Install the generated file(s) on this computer** and click **Finish**.

Figure 11. Installation Options Window



The procedure for using a `.inf` file to update the registry is Windows version dependant. To manually install a `.inf` file on any machine, including the one on which it was generated, open the appropriate `.inf` files in a text editor such as Windows Notepad and follow the instructions on the first few lines at the top.



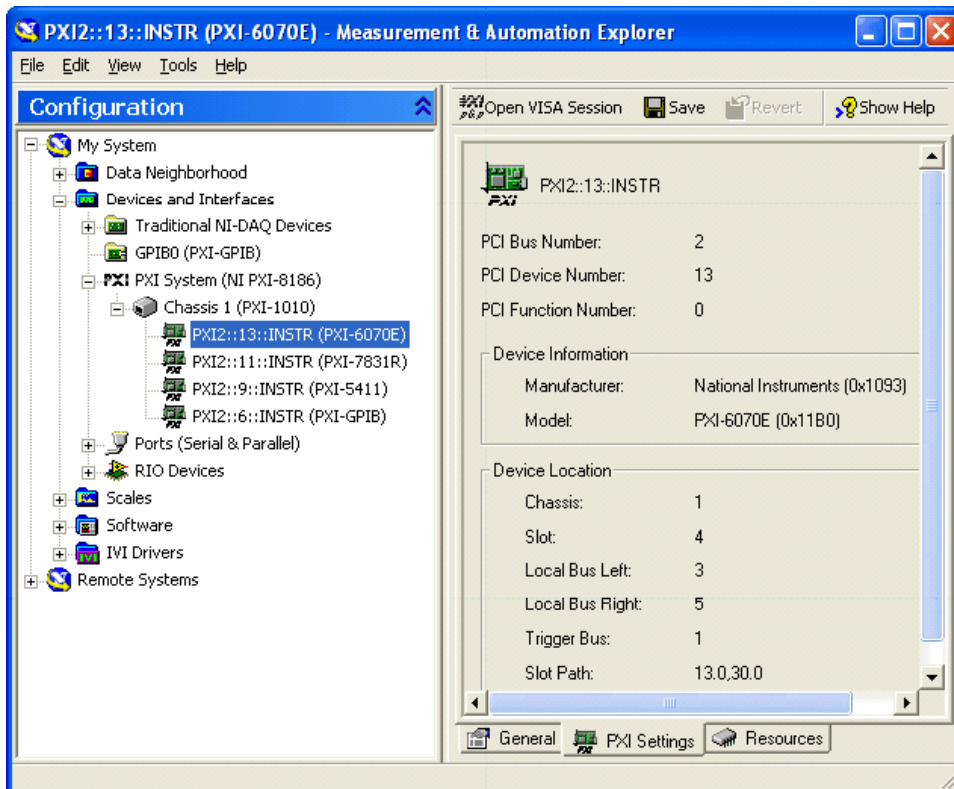
**Note** Choosing the **Install the generated file(s) on this computer** option requires administrator/root privilege. On Windows 10/8.1/7 SP1/Vista/Server 2008 R2, you will be prompted for credentials after choosing this option. On Linux, you must run the NI-VISA Driver Wizard as the root user for this to work properly. Otherwise, it will fail because you are not prompted for credentials at this point.



**Note** Only one driver should be associated with any particular PXI/PCI, or USB device in your system. If you are following the PXI-6070E example, for Windows to recognize NI-VISA as the driver for your device, you must make sure that NI-DAQ is not installed on the system. You also must be sure that any registry information about the old driver is removed.

You can now install your hardware in the system. Depending on your OS, this process may be slightly different. Once the hardware and setup information is properly installed, you should see the device through Measurement & Automation Explorer (MAX). Open MAX and press <F5> to refresh. The following figure shows what is visible in MAX after you properly install the `.inf` file you created using the wizard under Windows XP/Server 2003 R2 for your PXI-6070E.

Figure 12. PXI-6070E as an NI-VISA Device in MAX



**Note** If you have disabled the Passport for PXI devices in VISA, you must reenale the Passport for VISA to recognize PXI devices.

In the figure above, notice that several attributes are specified to the right of the PXI device listing in MAX. The NI-VISA Driver Wizard set the attributes listed under **Device Information** in the `.inf` file.

Now that the 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:

- The PCI bus number where it is located

- The PCI device number it is assigned
- The device function number

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 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 the figure above, the PXI device in this example has the resource name `PXI2::13::INSTR`. This name indicates that the PXI device is on PXI bus 2, assigned PXI device number 13, and 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, you can use the regular expression `"?* (PXI) ?*"` to tell the VISA driver to return only PXI device resources. You 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. 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, you can use VISA Interactive Control to develop the regular expression you need. The VISA Interactive Control is installed with VISA, and you can access it at **Start » Programs » National Instruments » VISA**. After opening the interactive control panel, 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 specify in your regular expression, and VISA Interactive Control creates the

regular expression for you.

### Related concepts:

- [Message-Based Communication Example Discussion](#)

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

Now that you can communicate with your device using NI-VISA, you will see a simple programming example to help you get started programming your device using the NI-VISA API. In the example, the PXI-6070E is programmed to toggle a few of its digital lines. The example demonstrates 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 LabWindows/CVI.

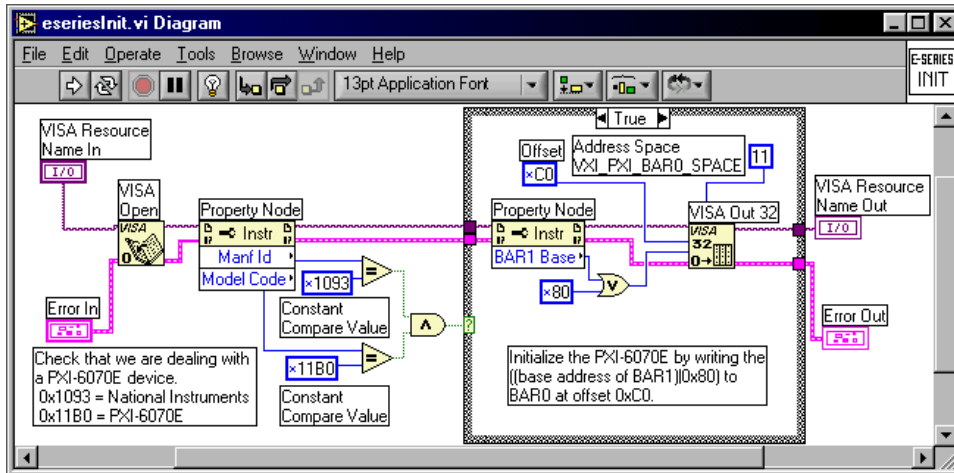
NI-VISA provides high-level and low-level register accesses. The valid address spaces for a PXI device are the configuration registers (VI\_PXI\_CFG\_SPACE) and the six Base Address Registers (VI\_PXI\_BAR0\_SPACE–VI\_PXI\_BAR5\_SPACE). A device may use any or all of the BARs. This information is device 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.

### Step 1 – 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. The following figure shows the diagram of the LabVIEW VI created to initialize the PXI-6070E.



Figure 13. PXI-6070E Example eseriesInit.vi



This VI opens a VISA session to a PXI-6070E by calling VISA Open in LabVIEW. In the system displayed previously in MAX, the VISA resource name PXI2::13::INSTR would be supplied to the **Instrument Handle** input of VISA Open. The corresponding text-based function is viOpen(). The following figure shows the LabWindows/CVI code to toggle the digital lines of the PXI-6070E via the text-based version of the NI-VISA API. Refer to the following code 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;

status = viClose (pxi6070E);
status = viClose (defaultRM);
return 0;
}

```

There is one key difference between the LabVIEW example and the LabWindows/CVI

example. The LabWindows/CVI example calls `viOpenDefaultRM`. You must call this function 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 the figure above is an optional input in the example. This input would be supplied if you used `eseriesInit.vi` 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.

## Step 2 – Communicating with the PXI Device

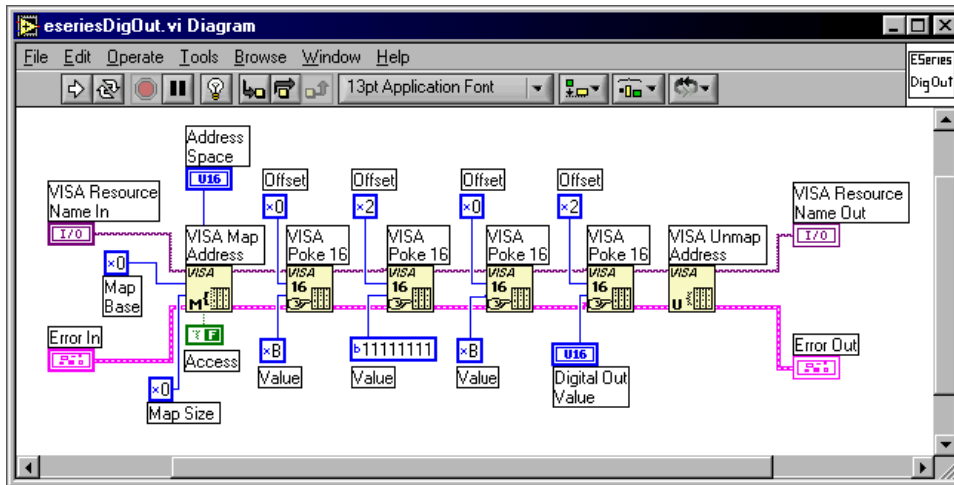
The final two steps in `eseriesInit.vi` are specific to the PXI-6070E initialization. A property node obtains the base address of the BAR1 register. This value is then compared with the value `0x80` using a logical OR. The result is written to BAR0 space at offset `0xC0`. To do this, you call the High-Level Register-Access function `VISA Out 32`, which writes a 32-bit value to the specified memory space at the specified offset. You use the decimal value 11 as an input to this VI to specify BAR0 as the address space to write to. As shown in the LabWindows/CVI example, 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 about using high-level and low-level functions to access registers on a device, refer to *Register-Based Communication*.

After you initialize and open a session to the PXI device, register peeks and pokes turn on a few of the digital lines on the device. The following figure shows 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 register values in the example are specific to the PXI-6070E, this example does 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, you must set up a hardware window 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 the example, the address space is BAR1, the offset is 0, and the size is 0x1000. Refer to the LabWindows/CVI example for a text-based example of this function.

Figure 14. eseriesDigOut.vi



After the window is properly mapped, the Poke function can write the appropriate values to the desired registers. Notice in the figure above 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 the LabWindows/CVI example. Here the example specifies the value to be written to the eight digital lines of the PXI-6070E. The binary value provided to the variable Digital Out Value is written to these lines. The preceding example shows only a few of the Low-Level-Register-Access VISA functions. Refer to Register-Based Communication for more information about Low-Level Register-Access functions.

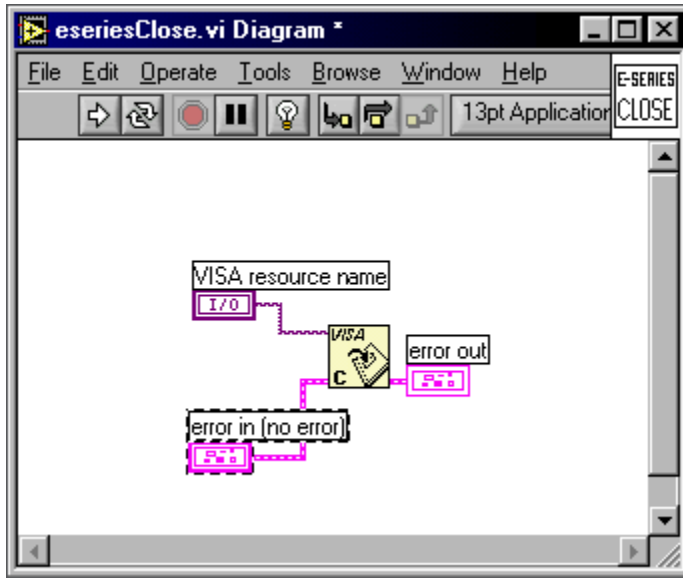
### Related concepts:

- [VISA Register-Based Communication](#)

## Step 3 – Closing the Device

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

Figure 15. Example eseriesClose.vi



In LabVIEW, to close a session to a VISA resource 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, 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.

## 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.

### 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, you must manually poll the event queue 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.

## 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 `viEnableEvent()` function. 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.

## 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 `viEnableEvent()` function 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 viDisableEvent() function 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, refer to Events.

# Programming Serial Devices in VISA

VISA supports programming Serial devices connected to either an RS-232 or RS-485 controller.

Serial users have traditionally faced difficulties when porting code from one platform to another. Each operating system has its own Serial API; each application development environment has its own Serial API; and all of these usually differ. The VISA Serial API is consistent across all supported platforms and all supported ADEs.

The first thing to point out is how to open a given Serial port. The format of the resource string that you pass to `viOpen()` is "ASRL<port>::INSTR". The actual binding of a given resource string to a physical port is platform dependent. Refer to the documentation and example in ***NI-VISA Platform-Specific and Portability Issues***. However, ASRL1::INSTR and ASRL2::INSTR are typically reserved for the native Serial ports (COM1 and COM2) on the local PC, if they exist.

VISA provides a consistent API across a broad range of Serial port controllers on all supported platforms. As operating systems continue to evolve and other new Serial APIs inevitably emerge, VISA will insulate you against unnecessary changes to your code.

## Related concepts:

- [NI-VISA Platform-Specific and Portability Issues](#)

## Default vs. Configured Communication Settings

When you open a Serial port, the VISA specification defines the default communication settings to be 9600 baud, 8 data bits, 1 stop bit, no parity, and no flow control. If you have configured the settings to a different value in the NI-VISA configuration utility (MAX on Windows, `visaconf` on UNIX), then you must pass the value `VI_LOAD_CONFIG (4)` as the `AccessMode` parameter to `viOpen()`. This parameter will cause the configured settings to be used; otherwise, if the `AccessMode` is 0 or `VI_NULL`, the default settings will be used.



Most Serial devices allow you to set the communication settings parameters via either DIP switches or via front panel selectors. If you are not using the NI-VISA configuration as discussed above, be sure to use `viSetAttribute()` to make these attribute values consistent with your device settings:

- `VI_ATTR_ASRL_BAUD` sets the baud rate. Defaults to 9600. The range depends on the serial port's capabilities and is platform dependent. For example, most but not all systems support 115200 baud.
- `VI_ATTR_ASRL_DATA_BITS` sets the number of data bits. Defaults to 8. The range is from 5–8.
- `VI_ATTR_ASRL_PARITY` sets the parity. Defaults to `VI_ASRL_PAR_NONE` (0). You can also choose odd, even, mark, or space.
- `VI_ATTR_ASRL_STOP_BITS` sets the number of stop bits. Defaults to `VI_ASRL_STOP_ONE` (10). Other valid values are `VI_ASRL_STOP_ONE5` (15) and `VI_ASRL_STOP_TWO` (20). Note that 1.5 stop bits is not supported on all systems and is also not supported in all combinations with other settings.
- `VI_ATTR_ASRL_FLOW_CNTRL` sets the method for limiting overflow on transfers between the devices. Defaults to `VI_ASRL_FLOW_NONE` (no flow control). You can also choose between XON/XOFF software flow control, RTS/CTS hardware flow control, and on supported systems, DTR/DSR hardware flow control.



**Note** DTR/DSR hardware flow control is not supported on Linux.

Other common (but not all) ASRL INSTR attributes are as follows:

- `VI_ATTR_ASRL_END_IN` defines the method of terminating reads. Defaults to `VI_ASRL_END_TERMCHAR`. This means that the read operation will stop whenever the character specified by `VI_ATTR_TERMCHAR` is encountered, regardless of the state of `VI_ATTR_TERMCHAR_EN`. To perform binary transfers (and to prevent VISA from stopping reads on the termination character) set this attribute to `VI_ASRL_END_NONE`.
- `VI_ATTR_ASRL_END_OUT` defines the method of terminating writes. Defaults to `VI_ASRL_END_NONE`. (This value means that the setting of `VI_ATTR_SEND_EN` is irrelevant.) To have VISA automatically append a termination character to each write operation, set this attribute to `VI_ASRL_END_TERMCHAR`. To have VISA automatically send a break condition after each write operation, set this attribute

to `VI_ASRL_END_BREAK`.

- If the serial port is RS-485, then you can query and manipulate the attribute `VI_ATTR_ASRL_WIRE_MODE`, which designates the RS-485 wiring mode. This attribute can have the values `VI_ASRL_WIRE4` (0, uses 4-wire mode), `VI_ASRL_WIRE2_DTR_ECHO` (1, uses 2-wire DTR mode controlled with echo), `VI_ASRL_WIRE2_DTR_CTRL` (2, uses 2-wire DTR mode controlled without echo), and `VI_ASRL_WIRE2_AUTO` (3, uses 2-wire auto mode controlled with TXRDY). This attribute is not supported for RS-232 ports. It is valid only on the platforms on which National Instruments supports RS-485 products.



**Note** DTR modes are not supported on Linux.

For lower-level functionality, you can also query the state of each modem line via `viGetAttribute()`. VISA will return whether the given line state is asserted (1), unasserted (0), or unknown (-1).

## Controlling the Serial I/O Buffers

The `viFlush()` and `viSetBuf()` operations also provide a control mechanism for the low-level serial driver buffers. The default size of these buffers is 0, which guarantees that all I/O is flushed on every access. To improve performance, you can alter the size of the low-level I/O transmit buffer or low-level I/O receive buffer by invoking the `viSetBuf()` operation with the `VI_IO_OUT_BUF` or `VI_IO_IN_BUF` flag, respectively. When the buffer size is non-zero, I/O to serial devices is not automatically flushed. You can force the low-level I/O transmit buffer to be flushed by invoking the `viFlush()` operation with `VI_IO_OUT_BUF`. Alternatively, you can call `viFlush()` with `VI_IO_OUT_BUF_DISCARD` to empty the low-level I/O transmit buffer without sending any remaining data to the device. You can also call `viFlush()` with either `VI_IO_IN_BUF` or `VI_IO_IN_BUF_DISCARD` to empty the low-level I/O receive buffer (both flags have the same effect and are provided only for API consistency).



**Note** Not all VISA implementations may support setting the size of either the low-level I/O receive or transmit buffers. In such an implementation, the `viSetBuf()` operation will return a warning. While this should not affect most programs, you can at least detect this lack of support if a specific buffer size

is required for performance reasons. If serial buffer control is not supported in a given implementation, we recommend that you use some form of handshaking (controlled via the `VI_ATTR_ASRL_FLOW_CNTRL` attribute), if possible, to avoid loss of data.

When using formatted I/O in conjunction with serial devices, calling `viFlush()` on a formatted I/O buffer has the same effect on the corresponding serial buffer. For example, invoking `viFlush()` with `VI_WRITE_BUF` flushes the formatted I/O output buffer first, and then the low-level I/O transmit buffer. Similarly, `VI_WRITE_BUF_DISCARD` empties the contents of both the formatted I/O and low-level I/O transmit buffers.



**Note** In previous versions of VISA, `VI_IO_IN_BUF` was known as `VI_ASRL_IN_BUF` and `VI_IO_OUT_BUF` was known as `VI_ASRL_OUT_BUF`.

#### Related concepts:

- [Automatically Flushing the Formatted I/O Buffers](#)
- [Formatted I/O Read and Low-Level I/O Receive Buffers](#)
- [Formatted I/O Write and Low-Level I/O Transmit Buffers](#)
- [Manually Flushing the Formatted I/O Buffers](#)
- [Recommendations for Using the VISA Buffers](#)

## National Instruments ENET Serial Controllers

The ENET to RS-232 and ENET to RS-485 products allow you to have the Serial controller box situated at a different location from your workstation. The workstation communicates over TCP/IP to the Serial controller box, which in turn communicates to the devices connected over the Serial bus. On most Windows operating systems, you can map each port on the controller box to a local port on the workstation, such as COM5.

NI-VISA currently natively supports communicating with these Serial controller boxes on Linux, Windows, LabVIEW RT, LabVIEW PDA, and Mac OS X. Because you cannot map the remote Serial ports to local Serial ports on the UNIX workstations, you must specify

the controller's hostname and the remote Serial port number directly in the resource string. This is also valid on Windows but is unnecessary if you have created a local Serial port mapping. The resource string for these products is

"ASRL::::<remote Serial port number>::INSTR". The hostname can be represented as either an IP address (dot-notation) or network machine name.

The communication settings discussion above applies to the ENET Serial controllers as well.

# Programming Ethernet Devices in VISA

VISA supports the discovery, identification, and programming of Ethernet devices over TCP/IP using the following methods:

- Raw socket connections
- LAN instrumentation protocol (also known as VXI-11)
- LAN eXtensions for Instrumentation (LXI) discovery/identification protocol

For users writing new code to communicate with an Ethernet instrument, the most important consideration in choosing the right API is which protocol(s) the device supports. The LAN instrument protocol, also known as VXI-11, was designed to mimic the message-based IEEE 488 style of programming with which instrumentation users have become accustomed; VISA is the best API to program devices using this protocol.

Some devices may be classified as LAN eXtensions for Instrumentation (LXI)-compatible devices. Many of these devices also use the LAN instrument communication protocol. However, some LXI devices may use a different communication protocol. For devices that have a published communication protocol, VISA is typically the best API to program these devices. If the protocol is not known or proprietary, a vendor-supplied instrument driver may be more appropriate.

For other devices, if the vendor merely documents the TCP/IP port number and proprietary raw packet format, VISA or any sockets API may be the best solution. Finally, some devices use other common well-defined protocols over either TCP/IP or UDP or some other layer; in these cases, an existing standard implementation of that protocol may be more appropriate than VISA.

For devices compatible with the LAN instrument protocol including most LXI devices, the simplest resource string is `"TCPIP::<hostname>::INSTR"`. The hostname can be represented as either an IP address (dot-notation) or network machine name. If an Ethernet device supports multiple internal device names or functions, you can access such a device with `"TCPIP::<hostname>::<device name>::INSTR"`. Recall that the `"INSTR"` resource class informs VISA that you are doing instrument (device) communication. Programming these LAN instruments in most cases is similar to programming GPIB instruments, in that most applications perform simple message-

based transfers (write command, read response) and receive service request event notifications. For more information about VISA message-based functionality, refer to ***Message-Based Communication***.

Newer LXI devices that implement HiSLIP 2.0, additionally support secure connections (TLS-based secure socket connections and SASL authentication mechanisms). These newer LXI devices can be accessed by specifying credential information in the VISA connection string "TCPIP::<credential\_id>@<hostname>::<device name>::INSTR". For details on secure LXI connections, refer to ***VISA Secure Connections***.

VISA provides a cross-platform API for programming Ethernet instruments. Other APIs provide the same Ethernet functionality and implement additional protocols, so if you are familiar with them, then there is not a strong reason for you to change to VISA. However, if you have instruments with more than one type of port or connection available to them (such as TCP/IP and GPIB on the same instrument), or are using multiple types of instruments with different hardware interface types, then using VISA may be advantageous because you can use the same interface independent API regardless of the connection medium. The only code that changes is the resource string.

#### Related concepts:

- [VISA Message-Based Communication](#)
- [VISA Secure Connections](#)

## VISA Sockets vs. Other Sockets APIs

For TCP/IP devices that you want to program directly (in the absence of a higher level protocol implementation), VISA provides a platform independent sockets API. VISA sockets are based on the UNIX sockets implementation in the Berkeley Software Distribution. A socket is a bi-directional communication endpoint, an object through which a VISA sockets application sends or receives packets of data across a network.

Additionally, VISA provides TLS 1.2+ support for programming devices that require a secure connection. For details on configuring secure connections, refer to ***VISA Secure Connections***.

The VISA socket resource string format is

"TCPIP::<hostname>::<port>::SOCKET". The SOCKET resource class informs VISA that you are communicating with an Ethernet device that does not support the LAN instrument or LXI protocol. By default, only the read and write operations are valid. If the device recognizes 488.2 commands such as "\*TRG\n" and "\*STB?\n", you can set the attribute VI\_ATTR\_IO\_PROT to VI\_PROT\_4882\_STRS (4) and then use the operations such as viAssertTrigger() and viReadSTB(). However, unlike LAN instruments, there is no way to support the service request event with the SOCKET resource class. For credential\_id configuration information, refer to **VISA Secure Connections**.

For users familiar with other platform independent sockets APIs, VISA does have some advantages. The VISA sockets API is simpler than the UNIX sockets API because viOpen() includes the functionality of socket(), bind(), and connect(). It is simpler and more portable than the Windows sockets API because it removes the need for calls to WSASStartup() and WSACleanup(). VISA uses platform independent VISA callbacks for asynchronous reads and writes so you do not need the platform specific knowledge of threading models and asynchronous completion services that other sockets APIs require. Finally, VISA is more powerful than that of many application development environments because it provides additional attributes for modifying the TCP/IP communication parameters.

The attribute VI\_ATTR\_TCPIP\_KEEPA\_LIVE defaults to false, but if enabled will use "keep-alive" packets to ensure that the connection has not been lost. The attribute VI\_ATTR\_TCPIP\_NODELAY defaults to true, which enforces that VISA write operations get flushed immediately; this ensures consistency with other supported VISA interfaces. The default setting disables the Nagle algorithm, which typically improves network performance by buffering send data until a full-size packet can be sent. Disabling this attribute (setting it to false) may improve the performance of multiple back-to-back VISA write operations to a TCP/IP device.

## Related concepts:

- [VISA Secure Connections](#)

## VISA Secure Connections

VISA provides TLS connectivity with TCPIP SOCKET type devices. Additionally, for TCPIP INSTR LXI devices, VISA supports SASL authentication mechanisms over TLS, as required by HiSLIP 2.0 devices.



**Note** Secure connections are currently only supported on Windows OS.



**Note** Enabling secure communication with the HiSLIP device will impact the throughput of the communication channel. During initialization, VISA will secure the connection to HiSLIP 2.0 devices and will allow toggling the security on or off afterwards by setting `VI_ATTR_TCPIP_HISLIP_ENCRYPTION_EN` at any point after the connection was secured successfully, as long as the device does not require mandatory encryption.



**Note** If enabled, NI-VISA Server provides remote access to machines that configure secure device connections. NI-VISA Server does not apply the TLS settings for remote connections to the VISA API.

## Configuring VISA Secure Connections

Secure connectivity is configured by adding "@" to the VISA resource string and, optionally, providing a credential identifier (such as "credential\_id@") that will be internally translated by NI-VISA to a user-password set or personal certificate, depending on the device type. Refer to ***VISA Resource Syntax and Examples*** for more information.

For TCPIP SOCKET type devices (`TCPIP[board]::[[credential_id]@]host address::port::SOCKET`), VISA supports the following credential strings:

- "@"—Enables TLS connections.
- "credential\_id@"—Enables TLS connections with client-side certification.

For TCPIP INSTR (HiSLIP 2.0) type devices



(TCPIP[board]::[[credential\_id]@]host address[:,HiSLIP device name[,HiSLIP port]][:,INSTR]), VISA supports the following credential strings:

- “@”—Sets up a TLS connection with ANONYMOUS SASL authentication.
- “credential\_id@”—Sets up a TLS connection with SCRAM-SHA256-PLUS, SCRAM-SHA256, or PLAIN SASL authentication—in this respective order of preference, from the most secure to the less secure option—depending on the SASL mechanisms supported by the instrument. In this case, the `credential_id` must point to a user-password set to be used during authentication. VISA will fall back to PLAIN SASL if the former mechanisms are not supported by the device and will always return an error if the provided user/password combination are not valid.

You can configure `credential_id` and trusted certificates by selecting **Tools » VISA Options** in the Hardware Configuration Utility application. The configuration window displays existing credentials and trusted certificates and allows add, edit, and remove operations.



**Note** The **Tools » VISA Options** menu is only available in the Hardware Configuration Utility if the installed VISA version supports secure connections.



**Note** The user and password strings, as well as personal certificates to be added as a `credential_id`, are device/user specific and information on creating and managing these is outside the scope of this document.




**Note** The storage of VISA credentials and certificates is specific to the NI-VISA implementation and not shared with other VISA implementations.

### Related concepts:

- [VISA Resource Syntax and Examples](#)

## Bypassing Secure Connection Certificate Errors

By default, VISA will refuse a secure connection with a server that authenticates using a certificate that is not trusted or has been revoked. In these cases, VISA will return `VI_ERROR_SERVER_CERT_UNTRUSTED` or `VI_ERROR_SERVER_CERT_REVOKED`. Using the following configuration tokens can change this behavior.

1. `AcceptUntrustedSslCertificates` token—located in the [TCPIP-RSRCS] section of `visaconf.ini`—can turn `VI_ERROR_SERVER_CERT_UNTRUSTED` into a warning, `VI_WARN_SERVER_CERT_UNTRUSTED`, by setting the following values:
    - 0 (default)—Secure connections can only be established with a trusted certificate. Otherwise, VISA will return `VI_ERROR_SERVER_CERT_UNTRUSTED`.
    - 1—A non-zero value (1) means that secure connections can be established even without a trusted server SSL certificate; in this case `viOpen` will return `VI_WARN_SERVER_CERT_UNTRUSTED` warning, and the connection will be established.
  2. `DisableSslCertificateRevocationCheck` token—located in the [TCPIP-RSRCS] section of `visaconf.ini`—configures certificate revocation check by setting the following values:
    - 0 (default)—VISA will apply the certificate revocation lists (CRL), already downloaded into one of the semicolon-separated directories pointed to by the `CRLDirectories` token from `visaconf.ini`, at the time `viOpen` is called. Accepted CRL file extensions are `.crl`, `.pem`, and `.der`.
- 

**Note** The directories referenced in `visaconf.ini` are user-created and should contain the CRLs to be disabled. Refer to the issuing Certificate Authority (CA) for information on your required CRLs.
- 1—A non-zero value (1) means that VISA will not apply CRLs when opening a connection.
  3. `AcceptIntermediateSslCertificatesAsTrustAnchors` token—located in the [TCPIP-RSRCS] section of `visaconf.ini`—configures certificate chain validation by setting the following values:

- 0 (default)—VISA will validate all the certificates from the server certificate chain and will fail the validation if the root certificate is missing from the VISA trusted certificates (see `AcceptUntrustedSslCertificates` for controlling whether this is returned as error or warning).
- 1—A non-zero value (1) means that VISA may accept an intermediate certificate as trust anchor—including self-signed certificates—if it is found in the trusted certificates. Note that when `DisableSslCertificateRevocationCheck=0`, certificate revocation is always verified, regardless of what value is set for `AcceptIntermediateSslCertificatesAsTrustAnchors`.

## Using Client-Side Certificates for VISA Secure Connections

For TCPIP SOCKET type devices, using client side certificates with VISA is possible for provided that the following steps are taken:

- The personal certificate is already in the current user's Windows Certificate Manager.
- A certificate based `<credential_id>` is defined to point to the previously stored personal certificate.
- The private key file is stored in the `[TCPIP-RSRCS]` section of the semicolon-separated directories defined in the `PrivateKeyDirectories` token of `visaconf.ini` and matches the "`<credential_id>.key`" filename.
- (Optional) A credential based `<credential_id>` is defined to store the passphrase for the private key file, as the password of `<credential_id>`.

## VISA Secure Connections Attributes

VISA Secure connections have additional attributes that can be queried to get details about the connection or device certificate. These attributes are:

- `VI_ATTR_TCPIP_SERVER_CERT`
- `VI_ATTR_TCPIP_SERVER_CERT_SIZE`
- `VI_ATTR_TCPIP_SERVER_CERT_ISSUER_NAME`
- `VI_ATTR_TCPIP_SERVER_CERT_SUBJECT_NAME`
- `VI_ATTR_TCPIP_SERVER_CERT_EXPIRATION_DATE`
- `VI_ATTR_TCPIP_SASL_MECHANISM`

- VI\_ATTR\_TCPIP\_TLS\_CIPHER\_SUITE
- VI\_ATTR\_TCPIP\_SERVER\_CERT\_IS\_PERPETUAL
- VI\_ATTR\_TCPIP\_SERVER\_CERT

Refer to ***INSTR*** and ***SOCKET*** for details on the available attributes.

#### Related information:

- [INSTR Resource](#)
- [SOCKET Resource](#)

# Programming Remote Devices in NI-VISA

NI-VISA allows you to programmatically access resources on a remote workstation. NI-DAQ users should find this similar to Remote DAQ.

Many users have devices that they need to use in multiple situations, such as a group of scientists sharing an instrument in the laboratory. The most common way this is done is for each user to physically carry the device next to his PC, connect the device, and then use it. NI-VISA for Windows and Linux now supports a more efficient way to do this. With remote NI-VISA on these supported platforms, you can leave the device connected to a single workstation and access it from multiple client workstations.

Remote NI-VISA is not a separate hardware interface type, but it is included in this help file for completeness.

Using remote NI-VISA is just one way to access hardware on another machine. If you have an existing application written using VISA and you need to use it from a different client, this may be the easiest solution. However, since each VISA operation invocation is a remote procedure call, your application performance may decrease, especially if it is register-intensive or has a significant amount of programming logic based on device responses or register values. The latency over Ethernet is better suited to applications that transfer large blocks of data. A better way to remotely access hardware is to make remote calls at a higher level, such as using Remote VI Server in LabVIEW.



**Note** NI-VISA Server does not restrict access to secure connections, nor does it apply the TLS settings for remoting the API.

## How to Configure and Use Remote NI-VISA

On the server machine (the one to which the hardware is connected), you must install Remote NI-VISA Server. This installation option may not exist in all NI-VISA distributions. By default, the server is disabled and access from any other computer is disallowed. Use the NI-VISA configuration utility (MAX on Windows, `visaconf` on UNIX) and make sure the server is enabled. You must also specify each address or address range of the computer(s) you wish to allow access. An address can be either in

dot notation (x.x.x.x) or the network machine name; an address range can only be in dot notation (x.x.x.\*).

On the client machine, no configuration is necessary. The VISA resource string contains the server machine name and the original VISA resource string on the server:

`"visa://hostname/VISA resource string"`. The hostname can be represented as either an IP address (dot-notation) or network machine name.

If you want to search for all resources on a specific server, you can pass

`"visa://hostname/?*" to viFindRsrc(). On Windows, you can use the Remote Servers section of MAX to configure NI-VISA to access certain servers by default. In this case, using "?*" will cause viFindRsrc() to query all configured servers as well as the local machine. If you want to limit the query to the local machine only, regardless of whether it has been configured to access any remote servers, pass "/?*" as the expression to viFindRsrc().`

Remote NI-VISA supports the complete functionality of all attributes, events, and operations for all supported hardware interface types.

# Programming USB Devices in VISA

NI-VISA supports programming USB devices using either low-level RAW access or the USB Test & Measurement Class protocol (also known as USBTMC).

For users writing new code to communicate with a USB instrument, the most important consideration in choosing the right API is which protocol(s) the device supports. The USB Test & Measurement Class (USBTMC) protocol was designed to mimic the message-based IEEE 488 style of programming with which instrumentation users have become accustomed; VISA is the best API to program devices using this protocol. For other devices, if the vendor merely documents the proprietary data format for each endpoint present on the device, VISA may be the best solution. Finally, some devices use other common well-defined protocols over USB; in these cases, an existing standard implementation of that protocol may be more appropriate than VISA. For example, operating systems ship standard drivers for certain USB device classes such as Human Interface Device (HID).

For RAW USB devices, NI-VISA supports communication on the default control pipe (endpoint 0) on all platforms. On some platforms, NI-VISA also supports nonzero endpoints. (Nonzero endpoints are not currently supported on Windows 7/Vista/Server 2008 R2.) It also supports bulk-in, bulk-out, interrupt-in, and interrupt-out communication. NI-VISA does not support communication on isochronous endpoints.

The best way to determine the resource string is to query the system with `viFindRsrc()` and use or display the resource(s) returned from that operation. Each USB device has a vendor code and a model code; this is much the same as PXI does, although the vendor IDs are different. You can create a query to search for devices of a particular attribute value; in this case, you can search for a specific vendor ID and model code. For example, the USB vendor ID for National Instruments is 0x3923. If NI made a device with the model code 0xBEEF, you could call `viFindRsrc()` with the expression `"USB?*{VI_ATTR_MANF_ID==0x3923 && VI_ATTR_MODEL_CODE==0xBEEF}"`. In many cases, the returned list has one or only a few devices.

NI-VISA provides a convenient means of accessing low-level RAW functionality of USB devices as well as support for the USB Test & Measurement Class (USBTMC) protocol.

The alternative to using NI-VISA for USB device communication is writing a kernel driver or using an existing kernel driver supplied by the vendor. By using NI-VISA, you avoid having to learn how to write kernel drivers, avoid having to learn a different kernel model for each operating system, and gain platform independence and portability by scaling to other operating systems in the future.

## Configuring NI-VISA to Recognize a RAW USB Device



**Note** NI-VISA will detect USB Test & Measurement Class (USBTMC) devices automatically. The information below applies only to setting up your USB device for low-level RAW access.

### Windows

#### Windows

Each USB device must have a kernel-level driver associated with it; this is done in Windows via a `.inf` file. For NI-VISA to recognize your device, you must run the NI-VISA Driver Wizard, available via the **Start » National Instruments » VISA** menu under .

The wizard first prompts you for basic information NI-VISA needs to properly locate your USB instrument. This includes the following:

- **USB Manufacturer ID**—This 16-bit value is vendor specific and is unique among USB-based device providers. It is referred to within the USB Specification as Vendor ID (VID). The vendor ID number for National Instruments, for example, is 0x3923.
- **USB Model Code**—The 16-bit device ID value is device specific, defined by the instrument provider, and required for USB-based devices. It is referred to within the USB Specification as Product ID (PID).
- **Compound Device**—Checking this box indicates that the USB device you are referring to is a compound device. A compound device is a device that has more than one USB interface. Each interface is much like a virtual device and may be controlled independently. You must specify the number of interfaces.

In text boxes where numerical information is required, preceding the number with 0x designates a hexadecimal value. The wizard assumes all other numeric entries are



decimal values.

The wizard also allows you to enter certain Windows Device Manager settings; these are cosmetic and do not affect the ability of NI-VISA to recognize and control your USB instrument. They are provided as a convenience, allowing you to more fully customize your instrument driver package.

When you are done, the NI-VISA Driver Wizard generates a Windows Setup Information (`.inf`) file for each supported operating system. Before a USB device will be visible to NI-VISA, you must use the `.inf` files to update the Windows system registry. The procedure for using a `.inf` file to update the registry is Windows-version dependent. To manually install a `.inf` file on any machine, including the one on which it was generated, open the appropriate `.inf` file in a text editor and follow the instructions on the first few lines at the top. Alternately, you can let the wizard install the `.inf` file appropriate for your machine before the wizard exits.

## Linux

NI-VISA relies on a Linux kernel feature for its USB support. This feature is called `usbfs`, and on older Linux kernels was referred to as ***usbdevfs***. For NI-VISA to support USB devices, this feature must be present and mounted (like a virtual filesystem). This is supported in most major Linux distributions such as Red Hat, SuSE, and Mandrake. You may use the `mount` command to display what filesystems are currently mounted to see if your system currently supports this feature.

Also, the VISA user must have write access to the file that represents the USB device, which is typically somewhere in a subdirectory within `/proc/bus/usb`. If this is not the case, the USB device is not accessible by VISA (it will not be found using `viFindRsrc`, and `viOpen` will fail). The default configuration on most systems is that the root user has write access; however, no other user has this access.

There are several options for providing a nonroot user access to a USB device.

- Use the hotplug package. This package is installed by default on most distributions including Red Hat, SuSE, and Mandrake. The hotplug package allows the user to run scripts for a specific USB device based on characteristics such as Vendor ID (VID) and Product ID (PID). If the hotplug package exists, the NI-VISA Installer by default will install scripts to give all users write access to all USB TMC devices and a

framework for USB RAW devices. To add write permissions for a specific USB RAW device, run the included script: `<VXIPNPPATH>/linux/NIvisa/USB/AddUsbRawPermissions.sh` For more information about the hotplug package, refer to the Linux Hotplugging website.

- `usbfs` (formerly known as `usbdevfs`) may be mounted with the option `devmode=0666`. This gives all users read and write access to all USB devices.
- The root user may add write permissions to the file that represents the USB device, which is typically somewhere in a subdirectory within `/proc/bus/usb`. Unfortunately, these permissions are lost if the device is unplugged. Therefore, this approach is not recommended.

## Mac OS X

As long as no other driver on the system claims the USB device, you can use NI-VISA to access it. No special setup is required.

## USB and VISA Background

VISA is a high-level application programming interface (API) for communicating with instrumentation buses. It is platform independent, bus independent, and environment independent. In other words, you use the same API regardless of whether you create a program to communicate with a USB device with LabVIEW on a machine running Windows XP, or a GPIB device with C on a machine running Mac OS X.

Universal Serial Bus (USB) is a message-based communication bus. This means a PC and USB device communicate by sending commands and data over the bus as text or binary data. Each USB device has its own command set. You can use NI-VISA Read and Write functions to send these commands to an instrument and read the response from an instrument. Check with your instrument manufacturer for a list of valid commands for your instrument.

NI-VISA supports USB communication. Two classes of VISA resources are supported: USB INSTR and USB RAW.

USB devices that conform to the USB Test and Measurement Class (USBTMC) protocol use the USB INSTR resource class. USBTMC devices conform to a protocol that the VISA USB INSTR resource class can understand. No configuration is necessary to

communicate with a USBTMC device. To communicate with a USBTMC instrument, refer to ***Using NI-VISA to Communicate with Your USB Device***. For more information about the USBTMC specification, refer to the USB Implementers Forum Web page.

USB RAW instruments are any USB instrument other than those instruments that specifically conform to the USBTMC specification. If you are using a USB RAW device, follow the instructions in ***Configuring NI-VISA to Control Your USB Device*** to configure NI-VISA to control your device. Contact your instrument manufacturer for details about the communication protocol and the command set your instrument uses.

#### Related concepts:

- [Using NI-VISA to Communicate with Your USB Device](#)
- [Configuring NI-VISA to Control Your USB Device](#)

## Using NI-VISA to Communicate with Your USB Device

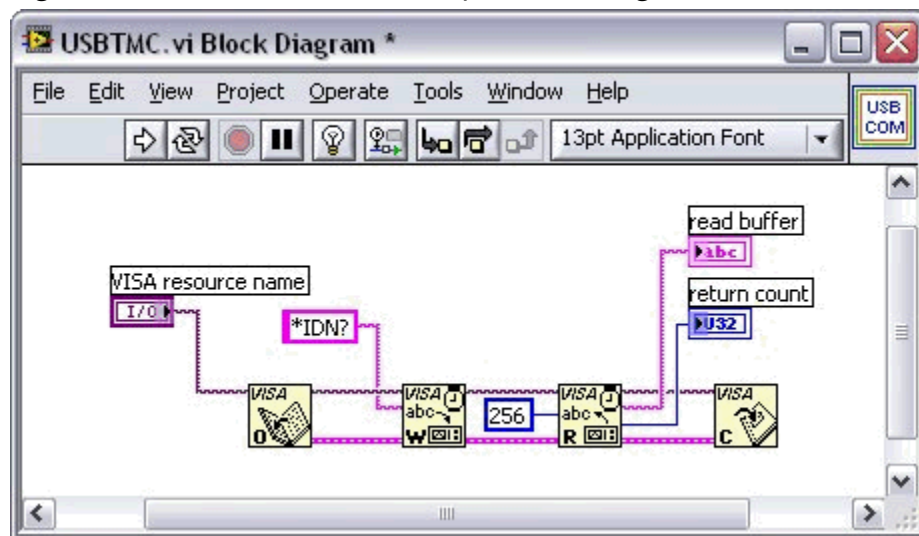
This topic explains how to communicate with your USB device using NI-VISA. Recall that there are two classes of USB devices. The communication method depends on the device class.

### USB INSTR Class (USBTMC)

Devices that conform to the USB Test and Measurement Class (USBTMC) use the NI-VISA USB INSTR class. These devices use 488.2-style communication. For these devices, use the VISA Open, VISA Close, VISA Read, and VISA Write functions the same way as if communicating with GPIB instruments.

The following figure shows a LabVIEW VI that communicates with a USBTMC device. In this example, the VI opens a VISA session to a USB device, writes a command to the device, and reads back the response. In this example, the specific command being sent is the device ID query. Check with your device manufacturer for your device command set. After all communication is complete, the VI closes the VISA session.

Figure 16. USBTMC LabVIEW Example Block Diagram



## USB RAW Class

Communicating with the USB RAW class is more complicated, because each device may use its own communication protocol. Contact your device vendor for details about the device communication protocol.

USB communicates using four types of pipes or endpoints: control, bulk, interrupt, and isochronous. Each type of pipe transfers a different type of information. Also, any number of endpoints can be of any endpoint type. Think of an endpoint as a communication socket. For specific details about USB architecture, review the USB Specification.

NI-VISA supports three types of USB pipes: control, bulk, and interrupt. When NI-VISA detects your USB instrument, it automatically scans your instrument for the lowest available endpoint for each type.

When VISA detects the lowest available endpoint, it assigns that value to the appropriate VISA attribute. The bulk in endpoint and bulk out endpoint are stored in the `VI_ATTR_USB_BULK_IN_PIPE` and `VI_ATTR_USB_BULK_OUT_PIPE` attribute, respectively. The interrupt in endpoint is stored in the `VI_ATTR_USB_INTR_IN_PIPE` attribute. A value of `-1` indicates a USB device does not support that type of pipe. For the control pipe, only endpoint zero is supported. If you are using the C API, use the `viSetAttribute` function to change endpoints. In LabVIEW, use a Write VISA Property Node.

NI-VISA includes four functions to transfer data through USB pipes. Before you can communicate with your device using these functions, you must set up the communication protocol using the VISA USB attributes. The following functions are available:

- Use VISA USB Control In and VISA USB Control Out to transfer data using the control pipe.
- To transfer data using a bulk pipe, use VISA Read and VISA Write.
- If you are using LabVIEW, VISA includes an additional function to use the interrupt pipe: VISA Get USB Interrupt Data. In the C API, you can do this by accessing the `VI_ATTR_USB_RECV_INTR_SIZE` and `VI_ATTR_USB_RECV_INTR_DATA` attributes of the `VI_EVENT_USB_INTR` event object.

To write an interrupt pipe out to a device, you first need to set the `VI_USB_BULK_OUT_PIPE` attribute in the C API or use the VISA USB Raw Out Pipe Property Node, shown below, in LabVIEW. Then, perform a VISA Write.

Figure 17. VISA USB Raw Out Pipe Property Node



## Configuring NI-VISA to Control Your USB Device

This topic explains how to configure a USB RAW device to be controlled by NI-VISA 3.0 on a Windows-based computer. If you are using a USBTMC-compatible device, connect your device and skip to Test Communication with NI-VISA Interactive Control.

At this point, NI-VISA should be installed on your computer, and your USB device should not be connected. Also, you should not have installed a driver for your USB device. There are three steps to configuring your USB device to use NI-VISA:

1. Create the `.inf` file using the NI-VISA Driver Wizard.
2. Install the `.inf` files and USB device.
3. Test communication with VISA Interactive Control.

This tutorial uses a NI DAQPad-6020E as an example USB device installed on a

Windows XP system. Because this tutorial explains how to configure of a generic USB device, it does not discuss details specific to the DAQPad-6020E. Remember that NI-DAQ is the only supported DAQPad-6020E driver.

### Related concepts:

- [Test Communication with VISA Interactive Control](#)
- [Configuring NI-VISA to Recognize a RAW USB Device](#)
- [Install the .inf Files and USB Device](#)

## Install the .inf Files and USB Device

The `.inf` file installation is different for each version of Windows. When the NI-VISA Driver Wizard creates a `.inf` file, installation instructions are included in a header at the top of the `.inf` file. Because `.inf` files are ASCII text files, any text editor such as Notepad can read them. For detailed information about installing your `.inf` file, open your `.inf` file in a text editor and follow the instructions at the top of the file.

To install the `.inf` files and USB device on Windows XP/Server 2003 R2, follow these steps:

1. Copy the `.inf` file to the `C:\Windows\inf` folder. This folder may be hidden, so you may need to change your folder options to view hidden files.
2. Right-click on the `.inf` file in `C:\Windows\inf` and click **Install**. This process creates a PNF file for your device. You are now ready to install your USB device.
3. Connect your USB device. Because USB is hot pluggable, Windows should detect your USB device, and the Add New Hardware Wizard should open automatically as soon as you connect your device to the USB port. Follow the onscreen instructions for the wizard. When you are prompted to select a driver for this device, browse to the `.inf` folder and select the `.inf` file you created using the driver wizard.

To install the `.inf` files and USB device on Windows 10/8.1/7 SP1/Vista/Server 2008 R2, follow these steps:

1. Right-click on the `.inf` file with the `vista` suffix (for example, `myusbdevice_vista.inf`) and click **Install**. This process creates a PNF file for your device. You are now ready to install your USB device.

2. Connect your USB device. Because USB is hot pluggable, Windows should detect your USB device, and the Found New Hardware Wizard should open automatically as soon as you connect your device to the USB port. Choose **Locate and Install Driver Software**. Follow the onscreen instructions for the wizard.

Proceed to the next step, Test Communication with VISA Interactive Control.

### Related concepts:

- [Test Communication with VISA Interactive Control](#)

## Test Communication with VISA Interactive Control

To test communication with VISA Interactive Control (VISAIC), follow these steps:

1. Open Measurement & Automation Explorer (MAX). Select **Tools » Refresh** to refresh the view. Your USB device should be listed under **Devices and Interfaces**. Your USB device is now installed and configured to use NI-VISA.

If you select your USB device, the **USB Settings** window displays the device information. You can use this information to access device information such as the manufacturer ID, model code, and serial number.

2. To communicate with your device using VISA, use the VISA instrument descriptor for your device. The instrument descriptor format for a USB INSTR device is `USB[board]:: manufacturer ID:: model code:: serial number[: USB interface number]:: INSTR`. The instrument descriptor format for a USB RAW device is `USB[board]:: manufacturer ID:: model code:: serial number[: USB interface number]:: RAW`.

According to the USBTMC specification, all USBTMC devices must have a serial number. Some USB RAW devices may not have serial numbers. If your device does not have a serial number, NI-VISA automatically assigns a VISA-specific serial number for the device. The serial number format is NI-VISA-#, where # is an automatically generated number.

Some USB devices have multiple interfaces. This is similar to the way a PCI device can have multiple functions. If your device supports only one interface, you do not

need to include the USB interface number.

The DAQPad-6020E uses the RAW class, and the manufacturer code and model code are 0x3923 and 0x12C0, respectively. For the DAQPad-6020E, the instrument descriptor is USB0::0x3923::0x12C0::00B50DAE::RAW.

To test communication with this device, open MAX. Select **Tools » NI-VISA » VISA Interactive Control**.

3. VISA Interactive Control (VISAIC) is a utility program for communicating easily with any VISA resource. After you configure your USB device to use VISA, it should be listed in the USB branch. Double-click on your device to open a VISA session to it.

When you open a VISA session with VISAIC, the Configuration page is displayed. The Configuration page contains all of the attributes that can be configured or set for the opened resource. The attributes are grouped together by category and separated into tabs. Open the View Attributes tab if you want to view all of the attributes for the selected resource.



# VISA Access Mechanisms

The following sections summarize the most important characteristics of attributes, events, and operations.

## Attributes

An attribute describes a value within a session or resource that reflects a characteristic of the operational state of the given object. These attributes are accessed through the following operations:

- `viGetAttribute()`
- `viSetAttribute()`

## Events

An event is an asynchronous occurrence that is independent of the normal sequential execution of the process running in a system. Depending on how you want to handle event occurrences, you can use the `viEnableEvent()` operation with either the `viInstallHandler()` operation or the `viWaitOnEvent()` operation.

Events respond to attributes in the same manner that resources do. Once your application is done using a particular event received via `viWaitOnEvent()`, it should call `viClose()` to destroy that event.

## Operations

An operation is an action defined by a resource that can be performed on the given resource. Each resource has the ability to define a series of operations. In addition to those defined by each resource, you can use the following template operations in any resource:

- `viClose()`
- `viGetAttribute()`
- `viSetAttribute()`
- `viStatusDesc()`

- viTerminate()
- viLock()
- viUnlock()
- viEnableEvent()
- viDisableEvent()
- viDiscardEvents()
- viWaitOnEvent()
- viInstallHandler()
- viUninstallHandler()

# VISA Resource Types

Currently, there are several VISA resource types—INSTR Resource, MEMACC Resource, INTFC Resource, BACKPLANE Resource, SERVANT Resource, SOCKET Resource, and RAW Resource. Most VISA applications and instrument drivers use only the INSTR resource.

## INSTR

A VISA Instrument Control (INSTR) resource lets a controller interact with the device associated with the given resource. This resource type grants the controller the following services to perform message-based and/or register-based I/O, depending on the type of device and the interface to which the device is connected.

Basic I/O services include the ability to send and receive blocks of data to and from the device. The meaning of the data is device dependent, and could be a message, command, or other binary encoded data. For devices compliant with IEEE 488, the basic I/O services also include triggering (both software and hardware), servicing requests, reading status bytes, and clearing the device.

Formatted I/O services provide both formatted and buffered I/O capabilities for data transfers to and from devices. The formatting capabilities include those specified by ANSI C, with extensions for common protocols used by instrumentation systems. Buffering improves system performance by making it possible to not only transfer large blocks of data, but also send several commands at one time.

Memory I/O (or Register I/O) services allow register-level access to devices connected to interfaces that support direct memory access, such as the VXIbus or VMEbus. Both high-level and low-level access services have operations for individual register accesses, with a trade-off between speed and complexity. The high-level access services also have operations for moving large blocks of data to and from devices. When using an INSTR resource, all address parameters are relative to the device's assigned memory base in the given address space; knowing a device's base address is neither required by nor relevant to the user.

Shared Memory services make it possible to allocate memory on a particular device

that is to be used exclusively by a given session. This is usually available only on devices that export shared memory specifically for such a purpose, such as a VXIbus or VMEbus controller.

## MEMACC

A VISA Memory Access (MEMACC) resource lets a controller interact with the interface associated with the given resource. Advanced users who need to perform memory accesses directly between multiple devices typically use the MEMACC resource. This resource type gives the controller the following services to access arbitrary registers or memory addresses on memory-mapped buses.

Memory I/O (or Register I/O) services allow register level access to interfaces that support direct memory access, such as the VXIbus or VMEbus. Both high-level and low-level access services have operations for individual register accesses, with a trade-off between speed and complexity. The high-level access services also have operations for moving large blocks of data to and from arbitrary addresses. When using a MEMACC resource, all address parameters are absolute within the given address space; knowing a device's base address is both required by and relevant to the user.

## INTFC

A VISA GPIB Bus Interface (INTFC) resource lets a controller interact with any devices connected to the board associated with the given resource. Advanced GPIB users who need to control multiple devices simultaneously or need to have multiple controllers in a single system typically use the INTFC resource. This resource type provides basic and formatted I/O services. In addition, the controller can directly query and manipulate specific lines on the bus, and also pass control to other devices with controller capability.

Basic I/O services include the ability to send and receive blocks of data onto and from the bus. The meaning of the data is device dependent, and could be a message, command, or other binary encoded data. The basic I/O services also include triggering devices on the bus and sending miscellaneous commands to any or all devices.

Formatted I/O services provide both formatted and buffered I/O capabilities for data transfers to and from devices. The formatting capabilities include those specified by

ANSI C, with extensions for common protocols used by instrumentation systems. Buffering improves system performance by making it possible to not only transfer large blocks of data, but also send several commands at one time.

## BACKPLANE

A VISA VXI Mainframe Backplane (BACKPLANE) resource encapsulates the operations and properties of each mainframe (or chassis) in a VXIbus system. This resource type lets a controller query and manipulate specific lines on a specific mainframe in a given VXI system. BACKPLANE services allow the user to map, unmap, assert, and receive hardware triggers, and also to assert and receive various utility and interrupt signals. This includes advanced functionality that might not be available in all implementations or on all controllers.

## SERVANT

A VISA Servant (SERVANT) resource encapsulates the operations and properties of the capabilities of a device and a device's view of the system in which it exists. The SERVANT resource exposes the device-side functionality of the device associated with the given resource. The SERVANT resource is a class for advanced users who want to write firmware code that exports message-based device functionality across potentially multiple interfaces. This resource type provides basic and formatted I/O services.

Basic I/O services include the ability to receive blocks of data from a commander and respond with blocks of data in return. The meaning of the data is device dependent, and could be a message, command, or other binary encoded data. The basic I/O services also include setting a 488-style status byte and receiving device clear and trigger events.

Formatted I/O services provide both formatted and buffered I/O capabilities for data transfers from and to the given device's commander. The formatting capabilities include those specified by ANSI C, with extensions for common protocols used by instrumentation systems. Buffering improves system performance by making it possible to not only transfer large blocks of data, but also send several commands at one time.

A VXI Servant resource also provides services to assert and receive various utility and interrupt signals.

## SOCKET

A VISA Ethernet Socket (SOCKET) resource encapsulates the operations and properties of the capabilities of a raw Ethernet connection using TCP/IP. The SOCKET resource exposes the capability of a raw socket connection over TCP/IP. This resource type provides basic and formatted I/O services.

Basic I/O services include the ability to send and receive blocks of data to and from the device. The meaning of the data is device dependent, and could be a message, command, or other binary encoded data. If the device is capable of communicating with 488.2-style strings, the basic I/O services also include software triggering, querying a 488-style status byte, and sending a device clear message.

Formatted I/O services provide both formatted and buffered I/O capabilities for data transfers to and from devices. The formatting capabilities include those specified by ANSI C, with extensions for common protocols used by instrumentation systems. Buffering improves system performance by making it possible to not only transfer large blocks of data, but also send several commands at one time.

## RAW

A VISA USB Raw (RAW) Resource encapsulates the operations and properties of the capabilities of a raw USB device. The RAW Resource exposes generic functionality of USB devices. This resource type provides basic and formatted I/O services.

Basic I/O services include the ability to send and receive blocks of data to and from the device. The meaning of the data is device dependent, and could be a message, command, or other binary encoded data. If the device is capable of communicating with 488.2-style strings, the basic I/O services also include software triggering, querying a 488-style status byte, and sending a device clear message.

Formatted I/O services provide both formatted and buffered I/O capabilities for data transfers to and from devices. The formatting capabilities include those specified by ANSI C, with extensions for common protocols used by instrumentation systems. Buffering improves system performance by making it possible to not only transfer

large blocks of data, but also send several commands at one time.

# VISA Resource Syntax and Examples

The following table shows the grammar for the address string. Optional string segments are shown in square brackets ([ ]).

Table 5. Grammar for Address Strings and Optional String Segments

| Interface            | Syntax   |
|----------------------|--|
| ENET-Serial<br>INSTR | ASRL[0]::host address::serial port::INSTR  |
| GPIB INSTR           | GPIB[board]::primary address [:: secondary address<br>] [::INSTR]                    |
| GPIB INTFC           | GPIB[ board ]::INTFC   |
| PXI<br>BACKPLANE     | PXI[interface]::chassis number::BACKPLANE  |
| PXI INSTR            | PXI[ bus ]:: device [:: function ] [::INSTR]   |
| PXI INSTR            | PXI[ interface ]:: bus-device [ .function ] [::INSTR]                                |
| PXI INSTR            | PXI[interface]::CHASSISchassis number::SLOTslot<br>number [::FUNCfunction] [::INSTR] |
| PXI<br>MEMACC        | PXI[ interface ]::MEMACC   |



| Interface      | Syntax  |
|----------------|---|
| Remote NI-VISA | <code>visa://host address[:server port]/remote resource</code>  |
| Serial INSTR   | <code>ASRLboard [::INSTR]</code>  |
| TCPIP INSTR    | <code>TCPIP[board]::host address [::LAN device name] [::INSTR]</code>   |
| TCPIP INSTR    | <code>TCPIP[board]::[[credential information]@host address [::HiSLIP device name [,HiSLIP port]] [::INSTR]</code> |
| TCPIP SOCKET   | <code>TCPIP[board]::host address :: port ::SOCKET</code>  |
| TCPIP SOCKET   | <code>TCPIP[board]::[[credential information]@host address :: port ::SOCKET</code>                                |
| USB INSTR      | <code>USB[board]::manufacturer ID::model code::serial number [::USB interface number] [::INSTR]</code>            |
| USB RAW        | <code>USB[board]::manufacturer ID::model code::serial number [::USB interface number] ::RAW</code>                |
| VXI BACKPLANE  | <code>VXI[board] [::VXI logical address] ::BACKPLANE</code>   |
| VXI INSTR      | <code>VXI[board]::VXI logical address [::INSTR]</code>  |

| Interface      | Syntax                   |
|----------------|--------------------------|
| VXI<br>MEMACC  | VXI [ board ] :: MEMACC  |
| VXI<br>SERVANT | VXI [ board ] :: SERVANT |

Use the GPIB keyword to establish communication with GPIB resources. Use the VXI keyword for VXI resources via embedded, MXIbus, or 1394 controllers. Use the ASRL keyword to establish communication with an asynchronous serial (such as RS-232 or RS-485) device. Use the PXI keyword for PXI and PCI resources. Use the TCPIP keyword for Ethernet communication.

Table 6. Optional String Segments

| Optional String Segments | Default Value |
|--------------------------|---------------|
| board                    | 0             |
| GPIB secondary address   | none          |
| LAN device name          | inst0         |
| HiSLIP device name       | hislip0       |
| HiSLIP port              | 4880          |
| PXI bus                  | 0             |

| Optional String Segments | Default Value                      |
|--------------------------|------------------------------------|
| PXI function             | 0                                  |
| credential information   | none                               |
| USB interface number     | lowest numbered relevant interface |

Table 7. Address Strings

| Address String          | Description  |
|-------------------------|--|
| ASRL::1.2.3.4::2::INSTR | A serial device attached to port 2 of the ENET Serial controller at address 1.2.3.4. |
| ASRL1::INSTR            | A serial device attached to interface ASRL1.   |
| GPIB::1::0::INSTR       | A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.      |
| GPIB2::INTFC            | Interface or raw board resource for GPIB interface 2.                                |
| PXI::15::INSTR          | PXI device number 15 on bus 0 with implied function 0.                               |
| PXI::2::BACKPLANE       | Backplane resource for chassis 2 on the default PXI system, which is                 |

| Address String                              | Description   |
|---|---|
|   | interface 0.  |
| PXI::CHASSIS1::SLOT3                        | PXI device in slot number 3 of the PXI chassis configured as chassis 1.   |
| PXI0::2-12.1::INSTR                         | PXI bus number 2, device 12 with function 1.  |
| PXI0::MEMACC                                | PXI MEMACC session.   |
| TCPIP::dev.company.com::INSTR               | A TCP/IP device using VXI-11 or LXI located at the specified address. This uses the default LAN Device Name of <code>inst0</code> .   |
| TCPIP0::@1.2.3.4::hislip0::INSTR            | A TCP/IP device using HiSLIP rev 2 located at IPv4 address 1.2.3.4. The connection is to be secured with an ANONYMOUS SASL authentication mechanism.  |
| TCPIP0::SecureCreds@1.2.3.4::hislip0::INSTR | A TCP/IP device using HiSLIP rev 2 located at IPv4 address 1.2.3.4. The connection is to be secured using the credential information (user, password) specified by <code>SecureCreds</code> . |
| TCPIP0::1.2.3.4::999::SOCKET                | Raw TCP/IP access to port 999 at the specified IP address.  |

| Address String                           | Description  |
|--|--|
| TCPIP0::SecureCreds@1.2.3.4::999::SOCKET | Raw TCP/IP access to port 999 at the specified IP address. The TLS connection is to be secured using the credential information (user certificate) specified by the string SecureCreds.        |
| TCPIP0::@1.2.3.4::999::SOCKET            | Raw TCP/IP access to port 999 at the specified IP address. The TLS connection is to be secured using only the server certificate validation.   |
| USB::0x1234::125::A22-5 ::INSTR          | A USB Test & Measurement class device with manufacturer ID 0x1234, model code 125, and serial number A22-5. This uses the device's first available USBTMC interface. This is usually number 0. |
| USB::0x5678::0x33::SN999::1::RAW         | A raw USB nonclass device with manufacturer ID 0x5678, model code 0x33, and serial number SN999. This uses the device's interface number 1.  |
| visa://hostname/ASRL1::INSTR             | The resource ASRL1 :: INSTR on the specified remote system.  |
| VXI::1::BACKPLANE                        | Mainframe resource for chassis 1 on the default VXI system, which is interface 0.  |

| Address String | Description  |
|----------------|--|
| VXI::MEMACC    | Board-level register access to the VXI interface.        |
| VXI0::1::INSTR | A VXI device at logical address 1 in VXI interface VXI0. |
| VXI0::SERVANT  | Servant/device-side resource for VXI interface 0.        |

**Related concepts:**

- [National Instruments ENET Serial Controllers](#)

# NI-VISA Platform-Specific and Portability Issues

The topics listed below discuss programming information for you to consider when developing applications that use the NI-VISA driver.

After installing the driver software, you can begin to develop your VISA application software. Remember that the NI-VISA driver relies on NI-488.2 and NI-VXI for driver-level I/O accesses.

- (Windows users) On VXI and MXI systems, use Measurement & Automation Explorer (MAX) to run the VXI Resource Manager (`resman`), configure your hardware, and assign VME addresses. For GPIB systems, use MAX to configure your GPIB controllers. To control instruments through Serial ports, you can use MAX to change the default settings, or you can perform all the necessary configuration at run time by setting VISA attributes.
- (All other platforms) On VXI and MXI systems, you must still run the VXI Resource Manager (`resman`) and use the VXI Resource Editor (`vxiedit` or `vxitedit`) for configuration purposes. For GPIB systems, you still use the GPIB Control Panel applet (Macintosh) or `ibconf` (UNIX) to configure your system. To control instruments through Serial ports, you can do all necessary configuration at run-time by setting VISA attributes. On UNIX, you can also use the VISA Configuration Utility (`visaconf`) to configure VISA aliases and change the default Serial settings.

## Related concepts:

- [NI-VISA Utilities](#)
- [Multiple Applications Using the NI-VISA Driver](#)
- [Low-Level Access Functions](#)
- [Interrupt Callback Handlers](#)
- [VXI and GPIB Platforms](#)
- [Serial Port Support](#)
- [VME Support](#)

## Multiple Applications Using the NI-VISA Driver

Multiple-application support is an important feature in all implementations of the NI-VISA driver. You can have several applications that use NI-VISA running simultaneously. You can even have multiple instances of the same application that uses the NI-VISA driver running simultaneously, if your application is designed for this. The NI-VISA operations perform in the same manner whether you have only one application or several applications (or several instances of an application) all trying to use the NI-VISA driver.

However, you need to be careful when you have multiple applications or sessions using the low-level bus access functions. The memory windows used to access the bus are a limited resource. Call the `viMapAddress()` operation before attempting to perform low-level bus access with `viPeekXX()` or `viPokeXX()`. Immediately after the accesses are completed, always call the `viUnmapAddress()` operation so that you free up the memory window for other applications.

## Low-Level Access Functions

The `viMapAddress()` operation returns a pointer for use with low-level access functions. On some systems, such as the VxIpc embedded computers, it is possible to directly dereference this pointer. However, on other systems, you must use the `viPeekXX()` and `viPokeXX()` operations. To make your source code portable between these and other platforms, and even other implementations of VISA, check the attribute `VI_ATTR_WIN_ACCESS` after calling `viMapAddress()`. If the value of that attribute is `VI_DEREF_ADDR`, you can safely dereference the address pointer directly. Otherwise, use the `viPeekXX()` and `viPokeXX()` operations to perform register I/O accesses.

National Instruments also provides macros for `viPeekXX()` and `viPokeXX()` on certain platforms. The C language macros automatically dereference the pointer whenever possible without calling the driver, which can substantially improve performance. Although the macros can increase performance only on NI-VISA, your application will be binary compatible with other implementations of VISA (the macros will just call the `viPeekXX()` and `viPokeXX()` operations). However, the macros are not enabled by default. To use the macros, you must define the symbol `NIVISA_PEEKPOKE` before including `visa.h`.



# Interrupt Callback Handlers

The way you register a handler depends on your programming language or development environment. Refer to the appropriate section below for how to register your handler.

After you install an interrupt handler and enable the appropriate event(s), an event occurrence causes VISA to invoke the callback. When VISA invokes an application callback, it does so in the correct application context. From within any handler, you can call back into the NI-VISA driver. On all platforms, you can also make system calls. The callback is performed in a separate thread created by NI-VISA. The thread is signaled as soon as the event occurs.

## C/C++

Application callbacks are available in C/C++ but not in LabVIEW or Visual Basic. Callbacks in C are registered with the `viInstallHandler()` operation and must be declared with the following signature:

```
ViStatus _VI_FUNCH appHandler (ViSession vi, ViEventType eventType, ViEvent event,
ViAddr userHandle)
```

Notice that the `_VI_FUNCH` modifier expands to `_stdcall` for Windows (32-bit). This is the standard Windows callback definition. On other systems, such as UNIX and Macintosh, VISA defines `_VI_FUNCH` to be nothing (null). Using `_VI_FUNCH` for handlers makes your source code portable to systems that need other modifiers (or none at all).

## Measurement Studio for Visual C++

When using National Instruments Measurement Studio for Visual C++, callbacks are registered with the `InstallEventHandler()` method. Refer to the Measurement Studio for Visual C++ documentation for more information on VISA callbacks. Handlers for this product must be declared with the following signature:

```
ViStatus __cdecl EventHandler (CNiVisaEvent& event)
```

## Measurement Studio for .NET

When using National Instruments Measurement Studio for .NET, you can register callbacks for specific events on different classes. For example, the `MessageBasedSession` class has a `ServiceRequest` event for which a callback can be registered. The code to register the callback for a `ServiceRequest` event on a `MessageBasedSession` in C# would look like the following code:

```
session.ServiceRequest += new MessageBasedSessionEventHandler(OnServiceRequest);
```

where `OnServiceRequest` would be declared as:

```
private void OnServiceRequest(object sender, MessageBasedSessionEventArgs e)
```

In VB .NET, the code to register a similar callback would look like:

```
AddHandler session.ServiceRequest, AddressOf OnServiceRequest
```

and `OnServiceRequest` would be defined as:

```
Private Sub OnServiceRequest(ByVal sender As Object, ByVal e As  
    ' Code here  
End Sub
```

Refer to the Measurement Studio for .NET documentation for more information.

## VXI and GPIB Platforms

NI-VISA supports all existing National Instruments GPIB, VXI, and Serial controllers for the operating systems on which NI-VISA exists. For VXI, this includes, but is not limited to, MXI-1, MXI-2, VXI-834x, VXI-1394, and the line of embedded VXIpc computers. For GPIB, this includes, but is not limited to, PCI-GPIB, GPIB-USB-A, AT-GPIB/TNT, PCMCIA-GPIB, and the GPIB-ENET and GPIB-ENET/100 boxes, which you can use to remotely control GPIB devices.

# Serial Port Support

The maximum number of serial ports that NI-VISA currently supports on any platform is 256. The default numbering of serial ports is system dependent, as shown in the following table.

Table 8. How Serial Ports Are Numbered

| Platform  | Method  |
|---|---|
| Windows 7/Vista/XP/<br>Server 2008 R2/Server<br>2003 R2 | All COM and LPT ports are automatically detected when you call viFindRsrc(). The VISA interface number may not equal the COM port number. |
| LabVIEW RT  | All COM ports are automatically detected when you call viFindRsrc().  |
| LabVIEW PDA   | All COM ports are automatically detected when you call viFindRsrc().  |
| Mac OS X  | All COM ports are automatically detected when you call viFindRsrc().  |
| Linux   | ASRL1-ASRL4 access /dev/ttyS0 – /dev/ttyS3.   |

If you need to know programmatically which ASRL INSTR resource maps to which underlying Serial port, the following code will retrieve and display that information.

## C Examples

```
#include "visa.h"
int main(void)
{
    ViStatus      status;                /* For checking errors */
    ViSession     defaultRM;             /* Communication channels */
    ViSession     instr;                 /* Communication channels */
    ViChar        rsrcName[VI_FIND_BUFLLEN]; /* Serial resource name */
    ViChar        intfDesc[VI_FIND_BUFLLEN]; /* Port binding description */
    ViUInt32      retCount;               /* To hold number of resources */
    ViFindList     flist;                 /* To hold list of resources */

    /* Begin by initializing the system */
    status = viOpenDefaultRM(&defaultRM);
    if (status < VI_SUCCESS) {

        /* Error Initializing VISA...exiting */
        return -1;
    }
}
```

```

}
status = viFindRsrc (defaultRM, "ASRL?*INSTR", &flist, &retCount, rsrcName);
while (retCount--) {
    status = viOpen (defaultRM, rsrcName, VI_NULL, VI_NULL, &instr);
    if (status < VI_SUCCESS)
        printf ("Could not open %s, status = 0x%08lX\n", rsrcName, status);
    else
    {
        status = viGetAttribute (instr, VI_ATTR_INTF_INST_NAME, intfDesc);
        printf ("Resource %s, Description %s\n", rsrcName, intfDesc);
        status = viClose (instr);
    }
    status = viFindNext (flist, rsrcName);
}
viClose (flist);
viClose (defaultRM);
return 0;
}

```

## Visual Basic Example

```

Private Declare Function viGetAttrString Lib "VISA32.DLL" Alias "#133" (ByVal vi As
ViSession, ByVal attrName As ViAttr, ByVal strValue As Any) As ViStatus

Private Sub vbMain()
    Dim stat          As ViStatus
    Dim dfltRM        As ViSession
    Dim sesn          As ViSession
    Dim fList         As ViFindList
    Dim rsrcName      As String * VI_FIND_BUFLen
    Dim instrDesc     As String * VI_FIND_BUFLen
    Dim nList         As Long
        stat = viOpenDefaultRM(dfltRM)
        If (stat < VI_SUCCESS) Then
            Rem Error initializing VISA ... exiting
            Exit Sub
        End If
        Rem Find all Serial instruments in the system
        stat = viFindRsrc(dfltRM, "ASRL?*INSTR", fList, nList, rsrcName)
        If (stat < VI_SUCCESS) Then
            Rem Error finding resources ... exiting
            viClose (dfltRM)
            Exit Sub
        End If
        While (nList)

```

```

stat = viOpen(dfltRM, rsrcName, VI_NULL, VI_NULL, sesn)
If (stat < VI_SUCCESS) Then
    Debug.Print "Could not open resource", rsrcName, "Status", stat
Else
    stat = viGetAttrString(sesn, VI_ATTR_INTF_INST_NAME, instrDesc)
    Debug.Print "Resource", rsrcName, "Description", instrDesc
    stat = viClose(sesn)
End If
stat = viFindNext(fList, rsrcName)
nList = nList - 1
Wend
stat = viClose(fList)
stat = viClose(dfltRM)
End Sub

```

## VME Support

To access VME devices in your system, you must configure NI-VXI to see these devices. Windows users can configure NI-VXI by using the Create New Wizard in MAX. Users on other platforms must use the Non-VXI Device Editor in VXI Resource Editor (vxiedit or vxitedit). For each address space in which your device has memory, you must create a separate pseudo-device entry with a logical address between 256 and 511. For example, a VME device with memory in both A24 and A32 spaces requires two entries. You can also specify which interrupt levels the device uses. VXI and VME devices cannot share interrupt levels. You can then access the device from NI-VISA just as you would a VXI device, by specifying the address space and the offset from the base at which you have configured it. NI-VISA support for VME devices includes the register access operations (both high-level and low-level) and the block-move operations, as well as the ability to receive interrupts.