

### NI-VISA<sup>™</sup> Help

June 2008, 370131L-01

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

For more information about this help file, refer to the following topics:

Using Help

**Related Documentation** 

<u>Glossary</u>

**Important Information** 

**Technical Support and Professional Services** 

To comment on National Instruments documentation, refer to the <u>National</u> <u>Instruments Web site</u>.

© 1996–2008 National Instruments Corporation. All rights reserved.

## **Related Documentation**

The following documents contain information that you may find helpful as you use this help file:

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*
- ANSI/IEEE Standard 488.2-1992, IEEE Standard Codes, Formats, Protocols, and Common Commands
- ANSI/IEEE Standard 1014-1987, IEEE Standard for a Versatile Backplane Bus: VMEbus
- ANSI/IEEE Standard 1155-1992, VMEbus Extensions for Instrumentation: VXIbus
- ANSI/ISO Standard 9899-1990, Programming Language C
- LXI, LAN eXtensions for Instrumentation, LXI Consortium
- *NI-488.2 Function Reference Manual for Windows*, National Instruments Corporation
- *NI-488.2 User Manual for Windows*, National Instruments Corporation
- *NI-VXI Help*, National Instruments Corporation
- *PXI Specification: PCI eXtensions for Instrumentation*, National Instruments Corporation
- VPP-1, Charter Document
- VPP-2, System Frameworks Specification
- VPP-3.1, Instrument Drivers Architecture and Design Specification
- VPP-3.2, Instrument Driver Developers Specification
- VPP-3.3, Instrument Driver Function Panel Specification
- VPP-4.3, The VISA Library
- VPP-4.3.2, VISA Implementation Specification for Textual Languages
- VPP-4.3.3, VISA Implementation Specification for the G Language
- VPP-5, VXI Component Knowledge Base Specification
- VPP-6, Installation and Packaging Specification
- VPP-7, Soft Front Panel Specification

- VPP-8, VXI Module/Mainframe to Receiver Interconnection
- VPP-9, Instrument Vendor Abbreviations
- VXI-11, TCP/IP Instrument Protocol, VXIbus Consortium

## **Using Help**

<u>Conventions</u> <u>Navigating Help</u> <u>Searching Help</u> <u>Printing Help File Topics</u>

# Conventions

This help file uses the following formatting and typographical conventions:

[]	Square brackets enclose optional items—for example, [response].
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence <b>File»Page Setup»Options</b> directs you to pull down the <b>File</b> menu, select the <b>Page Setup</b> item, and select <b>Options</b> from the last dialog box.
$\mathbf{N}$	This icon denotes a note, which alerts you to important information.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
<u>green</u>	Underlined text in this color denotes a link to a help topic, help file, or Web address.
italic	Italic text denotes variables, emphasis, cross–references, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.
monospace	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
monospace bold	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
monospace italic	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

# **Navigating Help (Windows Only)**

To navigate this help file, use the **Contents**, **Index**, and **Search** tabs to the left of this window or use the following toolbar buttons located above the tabs:

- **Hide**—Hides the navigation pane from view.
- Locate—Locates the currently displayed topic in the Contents tab, allowing you to view related topics.
- **Back**—Displays the previously viewed topic.
- Forward—Displays the topic you viewed before clicking the **Back** button.
- **Options**—Displays a list of commands and viewing options for the help file.

## **Searching Help (Windows Only)**

Use the **Search** tab to the left of this window to locate content in this help file. If you want to search for words in a certain order, such as "related documentation," add quotation marks around the search words as shown in the example. Searching for terms on the **Search** tab allows you to quickly locate specific information and information in topics that are not included on the **Contents** tab.

#### Wildcards

You also can search using asterisk (\*) or question mark (?) wildcards. Use the asterisk wildcard to return topics that contain a certain string. For example, a search for "prog\*" lists topics that contain the words "program," "programmatically," "progress," and so on.

Use the question mark wildcard as a substitute for a single character in a search term. For example, "?ext" lists topics that contain the words "next," "text," and so on.



**Note** Wildcard searching will not work on Simplified Chinese, Traditional Chinese, Japanese, and Korean systems.

#### **Nested Expressions**

Use nested expressions to combine searches to further refine a search. You can use Boolean expressions and wildcards in a nested expression. For example, "example AND (program OR VI)" lists topics that contain "example program" or "example VI." You cannot nest expressions more than five levels.

#### **Boolean Expressions**

Click the **•** button to add Boolean expressions to a search. The following Boolean operators are available:

- **AND** (default)—Returns topics that contain both search terms. You do not need to specify this operator unless you are using nested expressions.
- **OR**—Returns topics that contain either the first or second term.
- **NOT**—Returns topics that contain the first term without the second term.
- **NEAR**—Returns topics that contain both terms within eight words of each other.

#### **Search Options**

Use the following checkboxes on the **Search** tab to customize a search:

- **Search previous results**—Narrows the results from a search that returned too many topics. You must remove the checkmark from this checkbox to search all topics.
- Match similar words—Broadens a search to return topics that contain words similar to the search terms. For example, a search for "program" lists topics that include the words "programs," "programming," and so on.
- Search titles only—Searches only in the titles of topics.

# **Printing Help File Topics (Windows Only)**

Complete the following steps to print an entire book from the **Contents** tab:

- 1. Right-click the book.
- 2. Select **Print** from the shortcut menu to display the **Print Topics** dialog box.
- 3. Select the **Print the selected heading and all subtopics** option.
  - Note Select Print the selected topic if you want to print the single topic you have selected in the **Contents** tab.
- 4. Click the **OK** button.

#### **Printing PDF Documents**

This help file may contain links to PDF documents. To print PDF documents, click the print button located on the Adobe Acrobat Viewer toolbar.

### What You Need to Get Started

To use NI-VISA, you need the following:

- Appropriate hardware, in the form of a National Instruments GPIB, GPIB-VXI, MXI/VXI or serial interface board. For other hardware interfaces, the computer's standard ports should be sufficient for most applications.
- For GPIB applications, install NI-488.2. For VXI applications, install NI-VXI. For other hardware interfaces, NI-VISA uses the system's standard drivers.
- NI-VISA distribution media
- If you have a GPIB-VXI command module from another vendor, you need that vendor's GPIB-VXI VISA component. It will be installed into the <VXIPNPPATH>\<Framework>\bin directory. For example, the Hewlett-Packard component for the HPE1406 would be:

C:\Program Files\IVI Foundation\VISA\WinNT\bin\HPGPVX32.dll

## VXIplug&play Overview

The main objective of the VXI*plug&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 nextgeneration 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.

## **Supported Platforms**

This help file describes how to use NI-VISA, the National Instruments implementation of the VISA I/O standard, in any environment using LabWindows/CVI, any ANSI C compiler, or Microsoft Visual Basic. NI-VISA currently supports the frameworks and programming languages shown in the following table. 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.

	Windows Vista 64/ Vista 32/XP/2000	LabVIEW RT (Phar Lap ETS)	LabVIEW RT (VxWorks)	LabVIEW PD#
GPIB	1	4		
GPIB- VXI	4	4		
VXI	1			
Serial	1	1	4	1
PXI/PCI	1	1		
TCPIP	1	4	4	4
Remote Client	4	4		
Remote Server	4	4	4	
ENET Serial	4	4	4	4
USB	1		1	4
FireWire	1	1		

On Windows Vista/XP/2000, NI-VISA supports the WINNT and GWINNT frameworks defined by the VXI*plug&play* Systems Alliance and the IVI Foundation. There are no defined system frameworks on the other platforms.

The VXIplug&play Systems Alliance developed the concept of a

*framework* to categorize operating systems, programming languages, and I/O software libraries to bring the most useful products to the most end-users. A framework is a logical grouping of the choices that you face when designing a VXI system. You must always choose an operating system and a programming language along with an application development environment (ADE) when building a system. There are trade-offs associated with each of these decisions; many configurations are possible. The VXI*plug&play* Systems Alliance grouped the most popular operating systems, programming languages, and ADEs into distinct frameworks and defined in-depth specifications to guarantee interoperability of the components within each framework. To claim VXI*plug&play* compliance, a component must be compliant within a given framework.

With this version of NI-VISA, you can perform message-based and register-based communication with instruments, assert triggers, share memory, and respond to interrupts and triggers. For VXI, you can also perform register accesses at the interface level and mainframe-specific control and monitoring of utility lines. For GPIB, you also can perform board-level commands and the control and monitoring of bus lines. NI-VISA provides all the I/O functionality that you need for your test and measurement applications.

## How to Use the User Reference Help Topics

The User Reference help topics provide a sequential introduction to setting up a system to use VISA and then using and programming the environment. Please gather all the components described in <u>What You</u> <u>Need to Get Started</u>.

Once you have set up your system, you can use the <u>Introductory</u> <u>Programming Examples</u> topic to guide yourself through some simple examples. Other topics contain more in-depth information about the different elements that make up the VISA system.

For GPIB users or those familiar with NI-488.2, suggested reading is Introductory Programming Examples, Message-Based Communication, and GPIB Interface-Specific Information. For VXI users or those familiar with NI-VXI, suggested reading is Introductory Programming Examples, Register-Based Communication, and VXI Interface-Specific Information.

## **Introductory Programming Examples**

The following topics introduce some examples of common communication with instruments.

Example of Message-Based Communication

Example of Register-Based Communication

**Example of Handling Events** 

Example of Locking

To help you become comfortable with VISA, the examples avoid VISA terminology. <u>VISA Overview</u> looks at these examples again but using VISA terminology and focusing more on how they explain the VISA model.

These examples show C source code and Visual Basic syntax.

For Visual C# and Visual Basic .NET examples on Windows, go to the following directory: Program Files\National Instruments\MeasurementStudioVS2003\DotNET\Examples\Visa.

## **Example of Message-Based Communication**

Serial, GPIB, Ethernet, and VXI systems all have a definition of messagebased 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.

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

#### Example

```
#include "visa.h"
```

```
#define MAX_CNT 200
```

```
int main(void)
{
```

```
ViStatusstatus;/* For checking errors */ViSessiondefaultRM, instr;/* Communication channels */ViUInt32retCount;/* Return count from string I/O */ViCharbuffer[MAX_CNT];/* Buffer for string I/O */
```

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

```
/* 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;
```

```
}
```

Visual Basic Example Example Discussion

# Message-Based Communication Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 = 200Dim statAs ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionDim retCountAs LongDim bufferAs String \* MAX\_CNTRem Begin by initializing the systemstat = 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 example into the following steps.

- 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. <u>VISA Overview</u> has more details about viOpenDefaultRM(), but for now it is sufficient for you to understand that the function 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 GPIB*x* 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. For more information on address strings, viOpen(), and viOpenDefaultRM(), see Initializing Your VISA Application.

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.

For more information on these functions, see <u>Message-Based</u> <u>Communication</u>.

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 You can skip over this section if you are exclusively using GPIB, serial, or Ethernet communication. Register-based programming applies only to VXI, GPIB-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 <u>Low-Level Access Operations</u>.

#### Example

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, 8

```
/* 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;
```

```
}
```

Visual Basic Example

Example Discussion

# **Register-Based Communication Example (Visual Basic)**

Note The Visual Basic examples in the *NI-VISA Help* 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 statAs ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionDim deviceIDAs IntegerRem 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 this example is very similar to that of the <u>message-based communication example</u>. For this reason, we merely point out the basic differences as denoted in **bold** text:

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

The address string is still the same format as the address string in the <u>message-based communication example</u>, 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, such as from a GPIB-VXI to a MXIbus board.

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.

#### 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. If you are an experienced user or understand callback concepts, see VISA Events <u>Callbacks</u>.

#### 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.

#### Example

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;	
ViSession	defaultRM, instr;	
ViEvent	eventData;	
ViUInt16	statID;	

/\* For checking errors \*/
/\* Communication channels \*/
/\* To hold event info \*/
/\* Interrupt Status ID \*/

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

```
/* 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, &ii

```
/* Enable the driver to detect the interrupts */
status = viEnableEvent(instr, VI_EVENT_VXI_SIGP, VI_QUEUE, VI_]
```

```
/* 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) {</pre>
```

```
/* 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, &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; }
```

```
Visual Basic Example
Example Discussion
```

#### Handling Events Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 statAs ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionDim eTypeAs ViEventTypeDim eDataAs ViEventDim statIDAs IntegerRem Begin by initializing the systemstat = viOpenDefaultRM(dfltRM)If (stat < VI\_SUCCESS) Then</td>

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</pre>
```

```
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 interprocess 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.

#### Example

{

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

```
#include "visa.h"
```

```
#define MAX CNT 200
int main(void)
                                           /* For checking errors */
    ViStatus
                 status;
                                           /* Communication channels */
                 defaultRM, instr;
    ViSession
                                           /* Return count from string I/O */
    ViUInt32
                 retCount;
                                           /* Buffer for string I/O */
    ViChar
                 buffer[MAX CNT];
   /* 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, &ir
   /* 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_NI
   /* 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;

}

Visual Basic Example Example Discussion

## Locking Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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() and are discussed further in <u>VISA Locks</u>. 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 following topics give an overview of the VISA Library. Background Interactive Control of VISA VISA Terminology Beginning Terminology Communication Channels: Sessions The Resource Manager Examples of Interface Independence

# Background

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, GPIB-VXI, 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 VXI*plug&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**

Introductory Programming Examples introduced some examples of how to write code for the VISA driver. However, the topic deliberately avoided using VISA terminology to show that writing programs under VISA can be very straightforward and similar to software drivers you have used in the past. The following topics look at these examples again, but this time from the perspective of the underlying architecture.

**Beginning Terminology** 

Communication Channels: Sessions

The Resource Manager

# **Beginning 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 <u>VISA Resource</u> <u>Types</u>.

Returning to the <u>message-based communication example</u>, 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.

See <u>Message-Based Communication</u> for more information.

## **Communication Channels: Sessions**

The examples from Introductory Programming Examples 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 <u>message-based communication example</u>. 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.

#### The Resource Manager

The <u>Communication Channels: Sessions</u> topic briefly mentioned the VISA Resource known as the Resource Manager. What exactly 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 <u>message-based communication example</u>, 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().

# **Examples of Interface Independence**

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

First, 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.

A second example of interface independence is the GPIB-VXI controller. This lets you communicate with VXI devices, but through a GPIB cable. In other words, you use a GPIB interface with GPIB software to send messages to VXI devices, the same way you program stand-alone GPIB instruments. But how do you perform register accesses to the VXI devices? Prior to VISA, you were required to send messages to the GPIB-VXI itself and ask it to perform the register access. For example, when talking to the National Instruments GPIB-VXI/C with NI-488.2, the register access looks like the following when using NI-488.2 function calls:

dev = ibdev(boardID, PrimAddr, SecAddr, TMO, EOT, EOS);

status = ibwrt(dev, "A24 #h200000, #h1234", cnt);

If you had ever planned to move your code to a MXI or embedded VXI controller solution, you would spend a great deal of time changing your GPIB calls to VXI calls, especially when considering register accesses. VISA has been designed to eliminate problems such as this limitation. If you are talking to a VXI instrument, you can perform register I/O

regardless of whether you are connected via GPIB, MXI, or an embedded VXI computer. In addition, the code is exactly the same for all three cases. Therefore, the code for writing to the A24 register through a GPIB-VXI is:

status = viOut16(instr, VI\_A24\_SPACE, 0x0, 0x1234);

These examples show 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**

The following topics describe the steps required to prepare your application for communication with your device.

**Introduction** 

**Opening a Session** 

**Finding Resources** 

Finding VISA Resources Using Regular Expressions

Attribute-Based Resource Matching

Configuring a Session

Accessing Attributes

Common Considerations for Using Attributes

## Introduction

A powerful feature of VISA is the concept of a single interface for finding and accessing devices on various platforms. The VISA Resource Manager does this by exporting services for controlling and managing resources. These services include, but are not limited to, assigning unique resource addresses and unique resource IDs, locating resources, and creating 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.

## **Opening a Session**

When trying to access any of the VISA resources, the first step is to get a reference to the default Resource Manager by calling viOpenDefaultRM(). Your application can then use the session returned from this call to open sessions to resources controlled by that Resource Manager, as shown in the following example.

#### Example



**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#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, &in:
   /* Use device and eventually close it. */
   viClose(instr);
   viClose(defaultRM);
```

return 0;

}

As shown 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 <u>VISA Resource Syntax and</u> <u>Examples</u> for the syntax of resource strings and examples.

Refer to <u>NI-VISA Platform-Specific and Portability Issues</u> 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. In the Windows platform, COM1 corresponds to ASRL1, unlike in LabVIEW, where COM1

is accessible using port number 0.

The tables in <u>VISA Resource Syntax and Examples</u> show the canonical resource name formats. NI-VISA also supports the use of aliases to make opening devices easier. On Windows, run Measurement & Automation Explorer (MAX) and choose the menu option **Tools»NI-VISA»Alias Editor** to manage all your aliases. On UNIX, run visaconf and double-click any resource to bring up a dialog box for managing the alias for that resource. NI-VISA supports alias names that include letters, numbers, and underscores. To use an alias in your program, just call viOpen() with the alias name instead of the canonical resource name.

#### **Opening a Session Example (Visual Basic)**

Note The Visual Basic examples in the *NI-VISA Help* 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 statAs ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionRem Open Default RMstat = viOpenDefaultRM(dfltRM)If (stat < VI\_SUCCESS) Then</td>

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

# **Finding Resources**

As shown in the previous section, 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.

#### Example



**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

#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 autoConnect(ViPSession instrSesn)
{
    ViStatus
               status;
    ViSession defaultRM, instr;
    ViFindList fList;
               desc[VI_FIND_BUFLEN];
    ViChar
    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, des
   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;
```

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 that corresponding resource manager session. Therefore, 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().

#### Finding Resources Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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.

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

```
As ViStatus
Dim stat
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
   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.

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.
[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

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

	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.

Notice that in VISA, the string "GPIB?\*INSTR" applies to both GPIB and GPIB-VXI instruments.

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, GPIB1::1::INSTR, and GPIB-VXI1::8::INSTR.
GPIB[0-9]*::?*INSTR	Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not GPIB- VXI1::8::INSTR.
GPIB[^0]::?*INSTR	Matches GPIB1::1::1::INSTR but not GPIB0::2::INSTR or GPIB12::8::INSTR.
VXI?*INSTR	Matches VXI0::1::INSTR but not GPIB- VXI0::1::INSTR.
GPIB-VXI?*INSTR	Matches GPIB-VXI0::1::INSTR but not VXI0::1::INSTR.
?*VXI[0-9]*::?*INSTR	Matches VXI0::1::INSTR and GPIB- VXI0::1::INSTR.
ASRL[0-9]*::?*INSTR	Matches ASRL1::INSTR but not VXI0::5::INSTR.
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 and GPIB- VXI1::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 {VI_ATTR_GPIB_SECONDARY_ADDR > 0 && VI_ATTR_GPIB_SECONDARY_ADDR < 10}	Find all GPIB devices that have secondary addresses from 1 to 9.
ASRL? *INSTR{VI_ATTR_ASRL_BAUD == 9600}	Find all serial ports configured at 9600 baud.
?*VXI?INSTR{VI_ATTR_MANF_ID == 0xFF6 && !(VI_ATTR_VXI_LA ==0    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 is similar to the <u>finding resources example</u>, except that it 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.

#### Example

```
\overline{\mathbb{N}}
```

**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

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

```
#define MANF ID
                           0xFF6 /* 12-bit VXI manufacturer ID of device *
                           0x0FE /* 12-bit or 16-bit model code of device */
#define MODEL CODE
/* Find the first matching device and return a session to it */
ViStatus autoConnect2(ViPSession instrSesn)
{
    ViStatus status;
    ViSession defaultRM, instr;
              desc[VI_FIND_BUFLEN], regExToUse[VI_FIND_BUFLEN];
    ViChar
   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;
```

}

# Attribute-Based Resource Matching Example (Visual Basic)

This example uses functionality not available in Visual Basic. Refer to the <u>finding resources example</u> for sample code using viFindRsrc().

# **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** 

**Common Considerations for Using Attributes** 

## **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().

Refer to <u>Attributes</u> for a list of all available attributes you can use for each supported interface.

# **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 <u>Serial</u>.
- 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 <u>Interface-Specific Information</u> for attribute information relevant to each support hardware interface type.

### **Message-Based Communication**

The following topics show how to use the VISA library in message-based communication.

**Introduction** 

**Basic I/O Services** 

Synchronous Read/Write Services

Asynchronous Read/Write Services

**Clear Service** 

Trigger Service

Status/Service Request Service

Example VISA Message-Based Application

Formatted I/O Services

Formatted I/O Operations

I/O Buffer Operations

Variable List Operations

Manually Flushing the Formatted I/O Buffers

Automatically Flushing the Formatted I/O Buffers

Resizing 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

Formatted I/O Instrument Driver Examples

Integers

Floating Point Values

**Strings** 

Data Blocks

## Introduction

Whether you are using RS-232, GPIB, Ethernet, VXI, or USB, messagebased 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 <u>Basic I/O Services</u> and the <u>Formatted I/O Services</u> 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 help file discusses the uses of the Basic I/O Services and the Formatted I/O Services provided by the INSTR Resource in a VISA application.

## **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().

The following topics describe the operations provided by the VISA Instrument Control Resource for the Basic I/O Services.

Synchronous Read/Write Services Asynchronous Read/Write Services Clear Service Trigger Service Status/Service Request Service Example VISA Message-Based Application

# **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 <u>VISA Overview</u>, 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);

Refer to <u>Common Considerations for Using Attributes</u> for the default values of these attributes. Refer to <u>Attributes</u> for a complete list and description of the available 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 <u>VI\_EVENT\_IO\_COMPLETION</u>. For a more detailed example using asynchronous I/O, see the <u>queuing and callback mechanism example</u>.



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

# **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 \*/

# **Example VISA Message-Based Application**

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

#### Example



**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#include "visa.h"
```

```
int main(void)
{
```

```
ViSession
                     defaultRM, instr;
                     retCount;
ViUInt32
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;
```

}

#### **VISA Message-Based Application Example** (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionDim retCountAs LongDim idnResultAs String \* 72Dim resultBufferAs 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. See <u>Controlling the Serial I/O Buffers</u> for more information on that topic.

# **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 viVPrintf() and viVScanf(). 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 "Fx" 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:
```

2

```
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. Refer to <u>Operations</u> for more information.

## **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. See viFlush() for more information.

#### **Related Topics**

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 Topics**

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**

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 lowlevel I/O receive buffer. The formatted I/O read buffer eliminates the firstbyte 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 Topics**

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** 

VI\_ATTR\_RD\_BUF\_OPER\_MODE

VI\_ATTR\_WR\_BUF\_OPER\_MODE

<u>viFlush</u>

<u>viScanf</u>

<u>viSetBuf</u>

## 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. For more information about using the low-level I/O transmit buffer with serial resources, refer to <u>Controlling the Serial I/O Buffers</u>.



#### **Related Topics**

Automatically Flushing the Formatted I/O Buffers

Controlling the Serial 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 VI\_ATTR\_RD\_BUF\_OPER\_MODE VI\_ATTR\_WR\_BUF\_OPER\_MODE viFlush viFlush viPrintf
# **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. For details on these conditions, refer to <u>Automatically</u> <u>Flushing the Formatted I/O Buffers</u>.

#### **Related Topics**

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

<u>viFlush</u>

<u>viPrintf</u>

<u>viScanf</u>

<u>viSetBuf</u>

# **Formatted I/O Instrument Driver Examples**

The following topics show examples of VISA formatted I/O usage found in existing instrument drivers.

**Integers** 

**Floating Point Values** 

**Strings** 

Data Blocks

These topics 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 and l) 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

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

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

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.

#### Example

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

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.

#### Example

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.

#### Example

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

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, 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

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, %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 and l) 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
2	-32768 to +32767	0 to 65535

#### Examples

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);
```

# **Register-Based Communication**

The following topics show how to use the VISA library in register-based communication.

**Introduction** 

**High-Level Access Operations** 

**High-Level Block Operations** 

Low-Level Access Operations

**Overview of Register Accesses from Computers** 

Using VISA to Perform Low-Level Register Accesses

**Operations versus Pointer Dereference** 

Manipulating the Pointer

**Bus Errors** 

Comparison of High-Level and Low-Level Access

<u>Speed</u>

Ease of Use

Accessing Multiple Address Spaces

**Shared Memory Operations** 

Shared Memory Sample Code



**Note** You can skip these topics if you are using GPIB, Serial, Ethernet, or USB exclusively. Register-based programming applies only to PXI, VXI, and GPIB-VXI.

## Introduction

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(). For more information on this topic, refer to <u>Shared Memory Operations</u>.

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 space; knowing a device's base 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 highlevel, 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.

	<b>High-Level Access</b>	High-Level Block	Low-Level Access
Read	viIn8()	viMoveIn8()	viPeek8()
	viIn16()	viMoveIn16()	viPeek16()
	viIn32()	viMoveIn32()	viPeek32()
Write	viOut8()	viMoveOut8()	viPoke8()
	viOut16()	viMoveOut16()	viPoke16()
	viOut32()	viMoveOut32()	viPoke32()

**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 <u>register-based communication</u> <u>example</u>.

# **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.

## **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 National Instruments 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<sup>32</sup> 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.

### Example

 $\overline{\mathbb{N}}$ 

**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#include "visa.h"
```

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

```
int main(void)
```

{

```
/* For checking errors */
ViStatus
               status;
ViAddr
               defaultRM, instr;
                                         /* Communication channels */
                                        /* User pointer */
ViUInt16
               address;
                                        /* To store register value */
ViSession
               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, &ii
status = viMapAddress(instr, VI_A16_SPACE, 0, 0x40, VI_FALSE, VI_NU
&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;
```

}

## Manipulating the Pointer Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 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

## **Bus Errors**

The LLA operations do not report bus errors. In fact, viPeek*XX*() and viPoke*XX*() do not report any error conditions. However, the HLA operations do report bus errors. When using the LLA operations, you must ensure that the addresses you are accessing are valid.
# Comparison of High-Level and Low-Level Access

The following topics compare high-level and low-level access.

<u>Speed</u>

Ease of Use

Accessing Multiple Address Spaces

## 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 viMove*XX*() 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 <u>pointer manipulation example</u>. 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.

### Example



**Note** This example uses **bold text** to distinguish lines of code that are different from those in the <u>pointer manipulation example</u>.



**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#include "visa.h"
```

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

```
int main(void)
{
```

ViStatus	status;	/* For checking errors */
ViSession	defaultRM, self;	/* Communication channels */
ViAddr	address;	/* User pointer */
ViBusAddress	offset;	/* Shared memory offset */
ViUInt16	addrSpace;	/* Shared memory space */
ViUInt16	value;	/* To store register value */
	-	

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

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

}

```
/* Open communication with VXI Device at Logical Address 0 */
/* NOTE: For simplicity, we will not show error checking */
status = viOpen(defaultRM, "VXI0::0::INSTR", VI_NULL, VI_NULL, &:
```

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

```
/* Determine where the shared memory resides */
status = viGetAttribute(self, VI_ATTR_MEM_SPACE, &addrSpace);
```

status = viMapAddress(self, addrSpace, offset, 0x100, VI\_FALSE, VI\_NUI &address); viPeek16(self, address, &value);
/\* Access a different register by manipulating the pointer. \*/
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;
```

}

# Shared Memory Sample Code Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 statAs ViStatusDim dfltRMAs ViSessionDim selfAs ViSessionDim addrAs ViAddrDim offsAs LongDim mSpaceAs IntegerDim ValueAs IntegerRem 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
```

## **VISA Events**

The following topics describe the VISA event model and how to use it. They discuss the various events VISA supports and the event handling paradigm.

**Introduction** 

Supported Events

**Enabling and Disabling Events** 

Queuing

**Callbacks** 

Callback Modes

**Independent Queues** 

The userHandle Parameter

Queuing and Callback Mechanism Sample Code

The Life of the Event Context

Event Context with the Queuing Mechanism

Event Context with the Callback Mechanism

**Exception Handling** 

## Introduction

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. <u>Queuing</u> 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. <u>Callbacks</u> 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.

## **Supported Events**

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

#### 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 <u>Exception</u> <u>Handling</u> 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, 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 from the device.	Supported for PXI INSTR only. Not supported on all platforms.
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	Supported for Serial INSTR only. This event is

	Carrier Detect (DCD) line changed state.	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	Supported for Serial

	that at least one data byte has been received.	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 result in al 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_EI is ignored.

To learn about other event types defined for other resource classes, refer to <u>Interface-Specific Information</u> 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 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.

## **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);

 $\overline{\mathbb{N}}$ 

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 <u>handling events example</u> for an example of handling events via the queue mechanism.

## 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 <u>userHandle Parameter</u> 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 <u>Enabling and Disabling</u> <u>Events</u>.
- 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.

## **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.

## 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.

### Example



**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#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)
{
             jobID;
    ViJobId
    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, myCallba
bufferHandle);

/\* Enable the driver to detect events \*/
status = viEnableEvent(gpibSesn, VI\_EVENT\_SERVICE\_REQ, VI\_HNDL1
VI\_NULL);
status = viEnableEvent(gpibSesn, VI\_EVENT\_IO\_COMPLETION, VI\_QU1
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) {</pre>
```

```
/* 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;
```

}

### Queuing and Callback Mechanism Sample Code Example (Visual Basic)

**Note** The Visual Basic examples in the *NI-VISA Help* 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.

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_NU
```

```
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
```

## The Life of the 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**

The following topics describe how to use locks in VISA.

**Introduction** 

Lock Types

Lock Sharing

Acquiring an Exclusive Lock While Owning a Shared Lock

Nested Locks

Locking Sample Code

#### Introduction

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** 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).

#### Example

 $\overline{\mathbb{N}}$ 

**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

#include "visa.h"

#define MAX\_COUNT 128

```
int main(void)
```

{

ViStatus	status;
ViSession	defaultRM;
ViSession	instrIN,
ViChar	instrOUT;
ViByte	accKey[VI_FIND_BUFLEN];
ViEventType	buf[MAX_COUNT];
ViEvent	etype;
ViUInt32	event;
	retCount;

/\* For checking errors \*/ /\* Communication chani /\* Communication chani /\* Access key for lock \*/

- /\* To store device data \*
- /\* To identify event \*/
- /\* To hold event info \*/
- /\* To hold byte count \*/

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

```
/* 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, &in
status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, &in
```

/\* 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);
```

```
/* 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, &e
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;
```

```
}
```

#### Locking Sample Code Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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\_COUNT = 128

Dim stat	As ViStatus	
Dim dfltRM	As ViSession	
Dim sesnIN	As ViSession	
Dim sesnOUT	As ViSession	
Dim aKey	As String * VI_FIND_BUFLEN	
Dim buf	As String * MAX_COUNT	
Dim etype	As ViEventType	
Dim event	As ViEvent	
Dim retCount	As Long	
Rem Begin by ini	itializing the system	
stat = viOpenDefaultRM(dfltRM)		

'For checking errors 'Communication chan 'To store device data 'To store device data 'To identify event 'To hold event info 'To hold byte count

Rem Error initializing VISA...exiting Exit Sub

If (stat < VI\_SUCCESS) Then

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 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** dropdown list and follow the previous instructions.

## NI Spy: Debugging Tool

NI Spy 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 Spy highlights functions that return errors, so you can quickly determine which functions failed during your development. NI Spy 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 all platforms that support VISA, with the exception of Macintosh and VxWorks. 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.



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 on the **VISA I/O** tab.

The following figure shows the VISAIC opening window.

VISA Interactive Control	
e Edit View Help	
SA 1/0 Soft Front Panels NI 1/0	NATIONAL INSTRUMENTS © Copyright 1995-2007
🌾 View By Type 🧼 View By Co	nnection
My Computer Serial Instrument Resources ASRL2::INSTR ASRL10::INSTR ASRL10::INSTR PXI PXI Instrument Resources PXI PXI 2::0::INSTR P-PXI PXI 0::MEMACC	
Resources to Find:	
12	
Resource to Open:	
ASRL2::INSTR	

#### **VISAIC Opening Window**

The **Soft Front Panels** tab of the main VISAIC panel gives you the option to launch the soft front panels of any VXI*plug&play* instrument drivers that have been installed on the system.

The **NI I/O** tab gives you the option to launch the NI-VXI interactive utility or the NI-488.2 interactive utility. This gives you convenient links into the interactive utilities for the drivers VISA calls in case you would like to try debugging at this level.

The **Resources to Find** control in VISAIC is a string control for using VISA expressions to find specific instruments. Refer to <u>viFindRsrc</u> for information about creating custom expressions. You also can click the drop down arrow next to the control and select **Create Query** to open a dialog for specifying machines, interfaces, or attributes to filter on. The dialog populates the **Resources to Find** control with the appropriate text to limit your find to specific filtering conditions.

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 produces a window with a series of tabs for interactively running VISA commands. (This is the same as clicking the **Open Test Panel** button in Measurement & Automation Explorer (MAX).) The exact appearance of these tabs depends on which compatibility mode VISAIC is in. To access the compatibility mode and other VISAIC preferences, select **Edit**»**Preferences...** to bring up the following window.

Preferences	X
Preferences will be applied to r	new session windows.
Class Tab Color	Compatibility Mode
Operation Tab Color	LabWindows/CVI LabVIEW
	Default Resource View
Selected Tab Text Color	By Connection By Type
🔽 Show VISA session num	ber in window title
🗂 Show duplicate GPIB ar	nd GPIB-VXI resources
Load configured settings	: (when available)
<u>OK</u> Ca	ncel <u>D</u> efault

The VISA implementations are slightly different in LabVIEW and LabWindows/CVI. These differences are reflected in the operation tabs that are shown when you open a session to a resource.

• Windows users—VISAIC detects whether you have LabVIEW

and/or LabWindows/CVI installed on your system and sets the compatibility mode accordingly.

If you change the preferences, the new preferences take effect for any subsequent session you open.

When a session to a resource is opened interactively, a window similar to the following appears. This window uses the LabVIEW compatibility mode.

ASRL2::INSTR (Session 0x00B665F0)	×
Property Node (Set) Write Read Clear	Show All VISA Operations
Attribute Name	Current Value
Timeout Value 🗸 🗸 🗸	a 2000
	Return Status
View <u>All Settable Attributes</u>	×0.
Modify the value of the specified attribute.	<u> </u>

By default, the **Show All VISA Operations** checkbox unchecked, and only the most commonly used tabs are shown. By checking the **Show All VISA Operations** checkbox, you can use the more advanced features such as events and locks. The checkbox is checked in the following window.

ASRL2::INSTR (Session 0x00B681A0)	
Template Basic I/O Interface I/O	Show All VISA Operations
Enable Event Disable Event Discard Events Wait on E Property Node (Set) Property Node (Get) Lock Un	ivent llock
Attribute Name	Current Value
Timeout Value 🗸 🗸 🗸	a 2000
New Value D View <u>A</u> II Settable Attributes	Return Status

Several main tabs appear in the window. The initial tab is the **Template** tab, which contains all operations dealing with events, properties, and locks. Notice that there is a separate tab for each operation contained within the main tab. The other main tabs are **Basic I/O**, **Register I/O**, and **Interface I/O**. The **Basic I/O** tab contains the operations for message-based instruments, while the **Register I/O** tab contains the operations other busspecific operations. The **Register I/O** tab and **Interface I/O** tab are not displayed if they do not apply to the given bus or instrument type.

#### **Measurement & Automation Explorer**

Measurement & Automation Explorer (MAX) provides access to all National Instruments DAQ, GPIB, IMAQ, IVI, Motion, VISA, and VXI devices. With MAX, you can configure National Instruments 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.

For more information about using MAX, refer to the following topics:

NI-VISA Platform-Specific and Portability Issues

How to Configure and Use Remote NI-VISA

Additional Programming Issues

Default vs. Configured Communication Settings

Introduction to Programming GPIB Devices in VISA

**Opening a Session** 

VME Support

<u>viFindRsrc</u>

#### visaconf

visaconf is the VISA configuration utility for Linux and Mac OS X. For more information about using visaconf, refer to the following topics: Additional Programming Issues Default vs. Configured Communication Settings How to Configure and Use Remote NI-VISA NI-VISA Platform-Specific and Portability Issues Opening a Session viFindRsrc

#### **NI-VISA Driver Wizard Overview**

To make your PXI/PCI, USB, or FireWire 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 or FireWire device for use on all supported operating systems. For a USB device, the wizard generates two .inf files, one for Windows XP/2000, the other for Windows Vista. 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 Vista/XP/2000, LabVIEW RT, Linux, and Mac OS X. The .inf file created by the wizard can then be distributed with an instrument driver distribution kit.

#### Hardware Bus

This dialog allows you to select which hardware bus is used by the device you want to make visible to NI-VISA applications.

This wizard is not designed for use with devices that already have an installed device driver.

#### **Basic FireWire® Device Information**

This dialog contains basic information the operating system needs to locate and to associate your FireWire device with the NI-VISA driver software. This information includes essential hardware characteristics that uniquely specify the device.

Unit Specification ID

**Unit Software Version** 

Manufacturer Name

Model Name

#### **Basic PXI/PCI Device Information**

This dialog contains basic information the operating system needs to locate and to associate your PXI device with the NI-VISA driver software. This information includes essential hardware characteristics that uniquely specify the device, including module and manufacturer information.

Manufacturer Code Manufacturer Name Model Code Model Name Generates Interrupts Subsystem Manufacturer Code Subsystem Model Code Device Uses Subsystem This device uses PXI Express Load Settings from Module Description File

More about Module Description Files

#### **Basic USB Device Information**

This dialog contains basic information the operating system needs to locate and to associate your USB device with the NI-VISA driver software. This information includes essential hardware characteristics that uniquely specify the device, including module and manufacturer information.



**Note** Using this wizard may not be necessary. NI-VISA may already be able to detect your USB instrument if it conforms to the USB Test & Measurement Class (USBTMC) protocol. If this is the case, DO NOT use this wizard to create an additional .inf file.

Manufacturer Code Manufacturer Name Model Code

Model Name

**Compound Device** 

#### **Interrupt Detection Information**

Enabling PXI/PCI interrupt handling within an NI-VISA application is a two-step process. First, you must specify how your device detects a pending interrupt. Second, you must specify how to acknowledge a pending interrupt. The NI-VISA Driver Wizard will guide you through the process of enabling NI-VISA to perform these two steps.

Interrupt Detection Background

Add a step before

Add a step after

Edit a step

Remove a step

Select sequence

Add sequence

Remove sequence

#### **Interrupt Removal Information**

In addition to the Interrupt Detection sequence, NI-VISA also needs to know the sequence of register operations required to acknowledge a pending interrupt condition for your device. At interrupt time, if the NI-VISA driver determines that your device is asserting an interrupt (via the sequence of register accesses specified in the Interrupt Detection sequence), VISA will execute this Interrupt Removal sequence to quiet the pending interrupt.

This sequence of register operations is constructed using the same Read, Write, and Compare operations discussed in the previous step (Interrupt Detection). Individual register operations are entered in an identical manner.

Interrupt Removal Background

Add a step before Add a step after Edit a step Remove a step

#### **NI-VISA PXI Interrupt Information**

The **Interrupt Information** dialog allows you to describe an individual register access for the Interrupt Detection sequence or the Interrupt Acknowledge sequence.

Type of access

Address space

Compare mask

Width of access

Offset within space

Value to write or compare

#### **Disarm Interrupt Information**

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.

#### **PXI Express Configuration Information**

For PXI Express devices, NI-VISA must know the correct register operation sequence to obtain the slot the device is currently plugged into. After you insert a device into a PXI Express chassis and power on the device, the device reads the slot number from pins on the chassis and stores the number in a memory register. To configure the device and chassis automatically, this information must be read from the device memory. Use the NI-VISA Driver Wizard to define how to read this information.

The register operation sequence includes register reads, reads masked with a set value, and reads shifted a set number of times. Enter individual register operations into the sequence exactly the same way as interrupt detection, removal, and disarm sequences.

#### **Output Files Information**

The **Output Files** dialog gathers the remaining information the NI-VISA Driver Wizard needs to create the Setup Information (.inf) files. This information includes the instrument driver prefix for your device and the directory where the files should be saved.

**Output Files Background** 

Instrument Prefix

**Output File Directory** 

What do I do with the Output Files?

#### **Installation Options**

The **Installation Options** dialog offers the following choices for using the Setup Information (.inf) files after they are generated:

Install the Setup Information file

FTP to LabVIEW RT system

Go to folder

Do nothing more

#### 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.

This tutorial includes the following topics:

PXI and VISA Background

Configuring NI-VISA to Recognize a PXI/PCI Device

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

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

Using LabWindows/CVI to Install Your Device .inf Files

#### **USB Instrument Control Tutorial**

This tutorial is a starting point for using NI-VISA to communicate with a USB device. It is not intended as a starting point for learning about USB architecture or the various protocols used in USB communication. After reading this tutorial, you should be able to install a USB device and use NI-VISA to communicate with that device, as long as you understand the device communication protocol.

This tutorial includes the following sections:

USB and VISA Background

Configuring NI-VISA to Control Your USB Device

Using NI-VISA to Communicate with Your USB Device

USB on Linux and Mac

#### **Interface-Specific Information**

Although one of the benefits of VISA is an interface-independent API, there are times when you must understand the details of the specific interface with which you are working. The following topics provide additional information about each of the hardware interface types that NI-VISA currently supports.

<u>GPIB</u>

Introduction to Programming GPIB Devices in VISA

Comparison Between NI-VISA and NI-488.2 APIs

**Board-Level Programming** 

**GPIB Summary** 

<u>GPIB-VXI</u>

Introduction to Programming GPIB-VXI Devices in VISA

Register-Based Programming with the GPIB-VXI

Additional Programming Issues

**GPIB-VXI Summary** 

<u>VXI</u>

Introduction to Programming VXI Devices in VISA

VXI/VME Interrupts and Asynchronous Events in VISA

Performing Arbitrary Access to VXI Memory with VISA

Other VXI Resource Classes and VISA

Comparison Between NI-VISA and NI-VXI APIs

Summary of VXI in VISA

<u>PXI</u>

Introduction to Programming PXI Devices in NI-VISA

**User-Level Functionality** 

Configuring NI-VISA to Recognize a PXI Device

Using LabWindows/CVI to Install Your Device .inf Files

Other PXI Resource Classes and VISA

**PXI Summary** 

<u>Serial</u>

Introduction to Programming Serial Devices in VISA

Default vs. Configured Communication Settings

Controlling the Serial I/O Buffers

National Instruments ENET Serial Controllers

Serial Summary

Ethernet

Introduction to Programming Ethernet Devices in VISA

VISA Sockets vs. Other Sockets APIs

**Ethernet Summary** 

Remote NI-VISA

Introduction to Programming Remote Devices in NI-VISA

How to Configure and Use Remote NI-VISA

Remote NI-VISA Summary

<u>USB</u>

Introduction to Programming USB Devices in VISA

Configuring NI-VISA to Recognize a RAW USB Device

**USB Summary** 

USB Instrument Control Tutorial

FireWire

Introduction to Programming FireWire Devices in VISA

Configuring NI-VISA to Recognize an INSTR FireWire Device

FireWire Summary
# GPIB

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

Introduction to Programming GPIB Devices in VISA

Comparison Between NI-VISA and NI-488.2 APIs

**Board-Level Programming** 

**GPIB Summary** 

# Introduction to Programming GPIB Devices in VISA

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\_READDR\_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.

### **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.

C NI-488.2 Device Function	C VISA INSTR Operation	LabVIEW NI-488.2 Device Function	LabVIEW VISA INSTR Operation
ibdev	viOpen	<no equivalent=""></no>	VISA Open
ibonl	viClose	<no equivalent=""></no>	VISA Close
ibwrt	viWrite	GPIB Write	VISA Write
ibrd	viRead	GPIB Read	VISA Read
ibclr	viClear	GPIB Clear	VISA Clear
ibtrg	viAssertTrigger	GPIB Trigger	VISA Assert Trigger
ibrsp	viReadSTB	<del></del> GPIB Serial Poll	VISA Read STB
ibwait	viWaitOnEvent	Wait for GPIB RQS	₩SA® Wait for RQS
ibconfig	viSetAttribute	GPIB Initialization	Procerty VISA Property Node

NI-VISA and NI-488.2 Functions and Operations	<b>NI-VISA</b>	and NI-	488.2 Fur	nctions a	and O	peration
---	----------------	---------	-----------	-----------	-------	----------

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. As in the previous table, you can see that the APIs are almost identical.

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

#### **Board-Level Programming Functions and Operations**

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.

### **GPIB** Summary

Since both of these 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.

## **GPIB-VXI**

VISA supports programming VXI devices connected through a GPIB-VXI controller. The functionality is a subset of the VISA API for VXI devices connected through a native VXI controller.

Introduction to Programming GPIB-VXI Devices in VISA

Register-Based Programming with the GPIB-VXI

Additional Programming Issues

**GPIB-VXI Summary** 

# Introduction to Programming GPIB-VXI Devices in VISA

For new GPIB-VXI users, this controller makes VXI message-based devices appear as though they are GPIB devices with secondary addresses. This initially provided an easy transition into VXI for customers with existing GPIB systems, because they could use the same NI-488.2 API to control both types of instruments. However, this proved problematic for VXI register-based devices, because their addresses are not mapped directly into the GPIB system.

For controlling message-based VXI devices through a GPIB-VXI, the biggest difference between a program using NI-488.2 and one using VISA is in the calls made at the beginning and the end. For register-based devices, the differences are more significant. This section first discusses the basic changes common to both types of devices, then discusses some of the changes required for register-based programming.

For message-based programming, an NI-488.2 program would typically call ibdev() with the VXI device's primary and secondary GPIB addresses to get a handle to the specific device. In VISA, a program calls viOpen() with the VXI device's logical address (which is a more natural address because the device is VXI) to get a handle to it. The simplest and most common GPIB-VXI resource string is "GPIB-VXI::<logical address>::INSTR". Once you have a session to the VXI device, the NI-488.2 and VISA calls to communicate with the device are very similar, as covered above in the Comparison between NI-VISA and NI-488.2 APIs section.

## **Register-Based Programming with the GPIB-VXI**

Register-based programming does not have a straightforward mapping. Because register accesses using the GPIB-VXI involve sending requests to the controller itself (using the local command set), NI-488.2 programs would use ibdev() with the GPIB-VXI *controller's* primary and secondary GPIB addresses. In VISA, you call viOpen() with the VXI device's logical address, the same method for both message-based and register-based devices, and VISA handles sending the necessary messages to the controller. For programming the device, the following NI-488.2 messages and VISA operations are roughly equivalent:

NI-488.2 Message	VISA Operation
"Laddrs?" or "DLAD?"	viFindRsrc()
"RMentry?" or "DINF?"	viGetAttribute()
"Cmdr?"	viGetAttribute() with VI_ATTR_CMDR_LA
"LaSaddr?"	viGetAttribute() with VI_ATTR_GPIB_SECONDARY_ADDR
"Primary?"	viGetAttribute() with VI_ATTR_GPIB_PRIMARY_ADDR
"WREG" or "A16"	viOut16() with VI_A16_SPACE
"RREG?" or "A16?"	viIn16() with VI_A16_SPACE
"A24"	viOut16() with VI_A24_SPACE
"A24?"	viIn16() with VI_A24_SPACE
"SrcTrig"	viAssertTrigger()

#### **Register-Based Programming Messages and Operations**

Notice that with the INSTR register access operations viOut16() and viIn16(), you pass a device-relative offset in the specified address space. This is different from the GPIB-VXI/C local command set, which accepts absolute addresses. If your application currently uses absolute addressing and you do not want to convert to device-relative offsets, you may consider the MEMACC resource, which accepts absolute addressing. The form of the resource string for that class is "GPIB-

VXI<system>::MEMACC". You can also use the operations viOut8() and viIn8() to perform 8-bit accesses, which is not a feature supported by the local command set. VISA also defines 32-bit operations and accesses to A32 space, but because these are not implemented by the GPIB-VXI/C itself, they return errors.

If you have used the DMAmove code instrument in the past, you can instead use the viMoveInxx() and viMoveOutxx() operations instead. They make use of the GPIB-VXI's DMA functionality, but require only a single operation call, instead of the multiple calls required to send the command and data blocks and then poll waiting for the operation to complete. Using VISA to move blocks of data also means that you no longer need to load the DMAmove code instrument, as NI-VISA automatically downloads a separate code instrument to handle these and other operations.

### **Additional Programming Issues**

For advanced users, the GPIB-VXI Mainframe 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. The form of the resource string for this class is "GPIB-VXI<system>::BACKPLANE". Services in this resource class allow the user to map, unmap, and assert hardware triggers, and also to assert various utility signals.

Although the VISA API is almost identical for VXI and GPIB-VXI, the GPIB-VXI implements only a subset of this functionality. As mentioned above, the GPIB-VXI does not support 32-bit register accesses, nor does it support A32 space. The attributes VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, and VI\_ATTR\_WIN\_ACCESS\_PRIV can only be set to the value VI\_DATA\_PRIV; other address modifiers are not supported. The attributes VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, and VI\_ATTR\_WIN\_BYTE\_ORDER can only be set to the value VI\_BIG\_ENDIAN; little endian transfers are not supported. Also, while the GPIB-VXI does support service request events, it does not support receiving the following events: miscellaneous VXI signals or interrupts, triggers, SYSFAIL, or SYSRESET.

If you have more than one GPIB-VXI controller in your system, or if you change the primary address of a GPIB-VXI controller from its default (1 for the National Instruments GPIB-VXI/C), or if you have a GPIB-VXI controller from another vendor, then you need to configure NI-VISA to find such a controller. Use the NI-VISA configuration utility (MAX on Windows, visaconf on UNIX) and explicitly add a GPIB-VXI controller. You will be prompted for the GPIB controller number to which the GPIB-VXI is connected (usually 0), a unique GPIB-VXI controller number (which you are free to assign), and the primary and secondary addresses to which you have configured this GPIB-VXI controller.

#### **GPIB-VXI** Summary

In summary, using VISA to program VXI devices controlled by a GPIB-VXI is no different than if they are controlled with a native VXI controller such as the PCI-MXI-2 or a VXIpc. Although porting the code from NI-488.2 to VISA is not simple in the case of register-based programming, it will be code that is compatible with native VXI controllers.

# VXI

The following topics introduce you to the concepts of VXI (VME eXtensions for Instrumentation), VME, MXI (Multisystem eXtension Interface), and how you can control these buses using VISA.

Introduction to Programming VXI Devices in VISA VXI/VME Interrupts and Asynchronous Events in VISA Performing Arbitrary Access to VXI Memory with VISA

Other VXI Resource Classes and VISA

Comparison Between NI-VISA and NI-VXI APIs

Summary of VXI in VISA

# Introduction to 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. VXI 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 registerbased 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. These operations are discussed in more detail in Register-Based Communication.

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, VXI 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 messagebased 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. These operations are discussed in more detail in 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 <u>VISA Events</u>. 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.

# 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 is 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 National Instruments 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.

C NI-VXI Function	C VISA INSTR Operation	LabVIEW NI-VXI Function	LabVIEV VISA INSTR Operatio
InitVXIIibrary	viOpenDefaultRM, viOpen	InitVXIIibrary	VISA
CloseVXIIibrary	viClose	CloseVXIIibrary	VISA Class
WSwrt	viWrite	WSwrt WSwrt	VISA Write
WSrd	viRead	WSrd	VISA Read
WSclr	viClear	WSclr	VISA Clear

NI-VISA and NI-VXI Functions and Operation	NI-VISA	and NI-VXI	<b>Functions and</b>	d Operations
--	---------	------------	----------------------	--------------

WStrg, SrcTrig	viAssertTrigger	WStrg, SrcTrig	VISA Assert Trigger
VXlin, VXlout	vilnX, OutX	VXIIII, VXIout	VISA InX, VISAOut)
VXImove	viMoveInX, viMoveOutX	VXImove	VISA Move InX VISA Move OutX
MapVXIAddress	viMapAddress	MapVXIAddress	<mark>⊮</mark> ∰ VISA Maŗ Address
AssertVXlint	viAssertIntrSignal	AssertVXIint	VISA Assert Interrupt
EnableVXItoSignalInt	viEnableEvent	EnableVXItoSignalInt	VISA Enable Event
WaitForSignal	viWaitOnEvent	WaitForSignal	<mark>∛∡</mark> VISA Wai on Event
GetDevInfo	viGetAttribute	GetDevInfoLong	Property 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.

# Summary of VXI in VISA

Since the VISA API is very similar to the NI-VXI API, and both provide almost the same VXI functionality, which should you choose? National Instruments 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.

# PXI/PCI

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.

Introduction to Programming PXI Devices in NI-VISA

**User-Level Functionality** 

Configuring NI-VISA to Recognize a PXI Device

Using CVI to Install Your Device .inf Files

Other PXI Resource Classes and VISA

PXI Summary

#### Introduction to Programming PXI Devices in NI-VISA

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<br/>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.

### **User-Level Functionality**

An INSTR session to a PCI or PXI device provides the same registerlevel programming functionality as in VXI. NI-VISA supports both highlevel and low-level accesses, as discussed in <u>Register-Based</u> <u>Communication</u>. 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.

# 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** menu under **National Instruments»VISA**.

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

- Instrument Prefix—The VXI*plug&play* or IVI 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. The specification for these files is at <u>http://www.pxisa.org/Specifications.html</u>. 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, USB, and FireWire 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.

National Instruments 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 2000/XP and Windows Vista 32/Vista 64 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.

# **PXI Summary**

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.

# **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 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**.

The following topics explain how to configure NI-VISA to recognize a PXI/PCI device:

Hardware Bus

Basic Device Information

Interrupt Detection Information

Interrupt Removal Information

**Output Files Generation** 

**Installation Options** 

#### Hardware Bus

The NI-VISA Driver Wizard supports creating .inf files for PXI/PCI, USB, and FireWire devices. When the wizard opens, it displays the Hardware Bus window as shown in the following figure.

🧱 NI-VISA Dri	ver Wizard			
<u></u> ?н	lardware Bus Selection	on	×	NATIONAL INSTRUMENTS
Welcome I and creati by your d	to the VISA Driver Develop es the files to allow NI-VIS evice.	oment Wizard! This wiz A to control your devic	ard gathers the necessary info e. Please select the hardware	rmation bus used
Hardware	Bus			
	⊙ PXI/PCI	OUSB	◯ FireWire	
		< Back	Next > Cancel	Help

#### Hardware Bus Window

Because you want to control a PXI or a PCI device, select **PXI/PCI** and click **Next**. For more information about controlling a USB device, refer to the <u>USB Instrument Control Tutorial</u>.

**Basic Device Information** 

#### **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.

NI-VISA Driver Wizard					
PXI/PCI - Device Information					
Welcome to the VISA Driv information to allow NI-VI information about your de	er Development Wizard for PXI/PCI! 1 SA to control your PXI or PCI device. evice.	This wizard gathers the necessary Please enter the requested			
This wizard will generate Linux. The INF file tells th you specify here.	an INF file for use with Windows 2000/ ne operating system to allow NI-VISA t	XP, LabVIEW RT, Mac OS X, and to control the PXI or PCI device that			
Manufacturer ID (VID)	Subsystem Manufacturer ID	Manufacturer Name			
0x 1093	0000 Ox	National Instruments			
Model Code (PID)	del Code (PID) Subsystem Model Code Model Nam				
0x 11B0	0x 0000	PXI-6070E			
This device uses a	subsystem				
This device genera	ates interrupts				
This device uses P	XI Express				
		·			
Load Settings fro	m a Module Description File				
	< Back	Next > Cancel Help			

#### **Basic Device Information**

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 <u>Output Files Generation</u>.

**Interrupt Detection Information** 

## **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.

🚟 NI-VISA Driver Wizard	J					
PXI/PCI - Ir	terrupt Detec	tion				
Specify the steps to det (true) only if all 'Compar in each sequence.	Specify the steps to determine if your device is interrupting. Each sequence is considered successful (true) only if all 'Compare' steps match. There must be at least one sequence, and one 'Compare' step in each sequence.					
At least one sequence r belong to this device.	nust be considered	d successfu	ul (true) for the int	errupt to be de	termined to	
Sequence 0 💉			Add Sequence	Ren	nove Sequence	
Interrupt Detection Step	)width	Snace	Offset	Mask	Value	
Compare	Long (32 bits)	BARO	0x00000004	0x80000000	0x80000000	
Add Step Before Add Step After Edit Step Remove Step						
< Back Next > Cancel Help						

#### **Interrupt Detection Information**

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.

🛲 Interrupt Information	
Access Type	Access Width
Compare 💌	Long (32 bits) 🛛 💌
Address Space	Space Offset
BARO 💌	0x 0000001 <b>þ</b>
Compare Mask	Write/Compare Value
0x 80000000	0× 80000000
OK Can	cel Help

#### **Interrupt Information Window**

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** 

## **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.

쨺 NI-VISA Driver Wi	zard					
PXI/PCI	- Interrupt Rem	oval				
Specify the steps to condition from the F already read, no st	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.					
Interrupt Removal : Access Write	Steps Width Word (16 bits)	Space BAR0	Offset 0x00000010	Mask	Value 0xDE01	
Add Step Before     Add Step After     Edit Step     Remove Step       < Back						

#### **Interrupt Removal Information**

Just as recognizing an interrupt may take multiple steps, acknowledging an interrupt may also take multiple steps. To add a step, click the **Add a step before** or **Add a step after** button. The steps to remove the interrupt from your PXI/PCI device should be in the 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 16bit 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.

~
e Value

#### **Interrupt Information**

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** 

## **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.

🐖 NI-VISA Driver Wi	zard					
PXI/PCI	( - Interrupt Disa	rm				
Specify the steps t event of abnormal disarming interrupt	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 SI	teps					
Access	Width	Space	Offset	Mask	Value	
			00000014			
Add Step Before Add Step After Edit Step Remove Step						
< Back Next > Cancel Help						

#### **Interrupt Disarm Information**

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.

🎆 Interrupt Information	
Access Type	Access Width
Compare 🛛 👻	Long (32 bits)
Address Space	Space Offset
BARO 💌	0x 00000014
Compare Mask	Write/Compare Value
0x 8000000	0× 80000000
OK Can	cel Help

#### **Interrupt Information**

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** 

# **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.

🐨 NI-VISA Driver Wiza	ard					
🔋 PXI/PCI -	PXI Express Co	nfigurati	ion			
Specify the steps to configure it. There n	Specify the steps to obtain the PXI Express slot number of your device so that NI-VISA can properly configure it. There must be at least one step in the sequence in order to determine the Slot Number.					
PXI Express Slot Retr Access Read	rieval Steps Width Long (32 bits)	Space BAR0	Offset 0x00000014	Mask	Bits to Shift	
Add Step Before	Add Step Before     Add Step After     Edit Step     Remove Step       < Back					

#### **PXI Express Configuration Information**

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 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.

Access Type	Access Width
Read	Long (32 bits)
Address Space	Space Offset
BARO	✓ 0x 00000014
Read Mask	Bits to Shift
	0

#### **PXI Express Information**

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** 

## **Output Files Generation**

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

🚟 NI-VISA Driver Wizard	
Output Files Generation	
Thank you for using the VISA Driver Development Wizard! This wizard will now gener which allow NI-VISA to control your device. Please specify the INF file name prefix a directory in which to write the files.	rate the files Ind the
If you are creating a distribution kit for your driver or application to be installed on ot computers, you must copy the generated files into the appropriate directory. Instru doing this on a given operating system are included in the header of the generated I	ther Ictions for NF file.
Instrument Prefix (INF file name)	
PXI6070E	
Directory in which to save the generated files:	
C:\VXIpnp\WinNT\PXI6070E	Browse
Files to be generated:	
C:\VXIpnp\WinNT\PXI6070E\PXI6070E.inf C:\VXIpnp\WinNT\PXI6070E\PXI6070E.ini	
< Back Next > Cano	el Help

#### **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 windows appears as shown in next topic.

#### **Installation Options**

# **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.

🚟 NI-VISA Driver Wizard	
Installation Options	
The wizard has generated the files into the specified directory. You now have th files. Recall that the generated INF file contains operating system specific install.	e option to install those ation information.
• Install the generated files on this computer.	
○ FTP the INF file to a LabVIEW RT system.*	Transfer Files
• Take me to the folder containing the generated INI and INF files.	
<ul> <li>The FTP option is available only if you have LabVIEW RT installed AND you PXI/PCI hardware bus. If you choose this option, use the Transfer Files buringht to specify the FTP parameters and send the files.</li> </ul>	are using the utton at the
< Back Finish >	Cancel Help

#### **Installation Options**

N

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 Vista,

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, USB, or FireWire 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 for your PXI-6070E.



#### PXI-6070E as an NI-VISA Device in MAX

Note If you have disabled the Passport for PXI devices in VISA, you must reenable 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 <u>viOpen()</u> or VISA Open in LabVIEW to access a PCI or PXI device is "PXI<br/>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.

# 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. The functions in LabVIEW and LabWindows/CVI for obtaining VISA attributes are introduced in the following topics:

Step 1—Initialize the Device

Step 2—Communicating with the PXI Device

Step 3—Closing the Device

# **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 <u>viOpen()</u> function in a textbased 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.



#### 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 <u>viOpen()</u>. 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 <u>viOpen()</u> function in LabWindows/CVI.

#### LabWindows/CVI Example

#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, &]

// Obtain the manufacturer's ID and models code
status = viGetAttribute (pxi6070E, VI\_ATTR\_MANF\_ID, &manufacturerII
status = viGetAttribute (pxi6070E, VI\_ATTR\_MODEL\_CODE, &modelCol

// 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, &
```

```
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 textbased language, <u>viGetAttribute()</u> performs the same function. Step 2—Communicating with the PXI Device

## **Step 2—Communicating with the PXI Device**

The final two steps in <u>eseriesInit.vi</u> 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 <u>viGetAttribute</u> and <u>viOut32</u> 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 <u>Register-Based Communication</u>.

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 <u>viMapAddress()</u> 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.



#### 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 <u>viMapAddress()</u> must be passed to each <u>viPoke16</u> 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 <u>Register-Based Communication</u> for more information about Low-Level Register-Access functions.

Step 3—Closing the Device

# 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.



#### 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.

# 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 eventhandling 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 <u>viEnableEvent()</u> function. The function you use to associate a handler with an event is <u>viInstallHandler()</u>. After calling this function and then enabling events, the function you have specified using <u>viInstallHandler()</u> 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.

# Serial

VISA supports programming Serial devices connected to either an RS-232 or RS-485 controller.

Introduction to Programming Serial Devices in VISA

Default vs. Configured Communication Settings

Controlling the Serial I/O Buffers

National Instruments ENET Serial Controllers

Serial Summary

# Introduction to Programming Serial Devices in VISA

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.

# **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.

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 2wire 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.

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 Topics**

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 *x*86, 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::<hostname>::<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.

# **Serial Summary**

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.
## Ethernet

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

Introduction to Programming Ethernet Devices in VISA VISA Sockets vs. Other Sockets APIs

Ethernet Summary

# Introduction to Programming Ethernet Devices in VISA

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 welldefined 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.

## **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.

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 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 WSAStartup() 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\_KEEPALIVE 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.

## **Ethernet Summary**

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.

## **Remote NI-VISA**

NI-VISA allows you to programmatically access resources on a remote workstation. NI-DAQ users should find this similar to Remote DAQ.

Introduction to Programming Remote Devices in NI-VISA

How to Configure and Use Remote NI-VISA

Remote NI-VISA Summary

# Introduction to Programming Remote Devices in NI-VISA

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 *x*86 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.

## 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.

## **Remote NI-VISA Summary**

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.

## USB

NI-VISA supports programming USB devices using either low-level RAW access or the USB Test & Measurement Class protocol (also known as USBTMC).

Introduction to Programming USB Devices in VISA

Configuring NI-VISA to Recognize a RAW USB Device

**USB Summary** 

**USB Instrument Control Tutorial** 

# Introduction to Programming USB Devices in VISA

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 Vista.) It also supports bulk-in, bulk-out, interruptin, 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.

# 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

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** menu under **National Instruments»VISA**.

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 following Web site: <u>http://linux-hotplug.sourceforge.net/</u>.

- 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 Summary**

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.

## **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 <u>Configuring NI-VISA to Control Your USB Device</u> 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.

## **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 <u>Test</u> <u>Communication with NI-VISA Interactive Control</u>.

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. <u>Create the .inf file using the NI-VISA Driver Wizard.</u>
- 2. Install the .inf files and USB device.
- 3. <u>Test communication with VISA Interactive Control.</u>

This tutorial uses a National Instruments 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.

# Create the .inf File Using the NI-VISA Driver Wizard

To use NI-VISA, you must first tell Windows to use NI-VISA as default driver for the device. In the Windows environment, you can do this with a .inf file. NI-VISA includes the NI-VISA Driver Wizard to create a .inf file for your USB device.

1. To open the DDW, run the NI-VISA Driver Wizard, available via the **Start** menu under **National Instruments»VISA**. The following figure shows the open screen.

🐺 NI-VISA Drive	r Wizard			
Har	dware Bus Selection	on		
Welcome to t and creates by your devi	the VISA Driver Develop the files to allow NI-VIS ce.	oment Wizard! This wi A to control your devi	zard gathers the necessa ce. Please select the har	ary information dware bus used
Hardware Bu	IS			
	○ PXI/PCI	⊙ USB	◯ FireWire	
		< Back	Next > Ca	ncei Help

#### VISA DDW Hardware Bus Window

You can use this wizard to create a .inf file for a PXI/PCI, USB, or FireWire device. Because you are creating the driver for a USB device, click **USB** and **Next**. The VISA DDW Basic Device Information window opens as shown in the following figure.

USB - Device Information		
Welcome to the VISA Driver Development information to allow NI-VISA to control yo about your device.	: Wizard for USB! This wizard gathers t ur USB device. Please enter the reque	he necessary sted information
This wizard will generate an INF file for us system to allow NI-VISA to control the US	e with Windows 2000/XP. The INF file I B device that you specify here.	tells the operating
USB Manufacturer ID (Vendor ID)	Manufacturer Name	
0x 3923	National Instruments	
USB Model Code (Product ID)	Model Name	
0x 12C0	DAQPad-6020E	
Compound Device?	2 Number of Interface:	s
NOTE: Using this wizard may NOT be nec Measurement Class (USBTMC) protocol, t	essary! If your device conforms to the hen NI-VISA can already detect and co	USB Test & ntrol it.In this
case you should not use this wizard to cre	eate an additional INF file.	
	< Back Next >	Cancel Help

#### **VISA DDW Basic Device Information Window**

2. For this step, you must know the USB vendor ID and product ID for your USB instrument. These numbers identify your USB device when you install it and address your device when you want to communicate with it. According to the USB specification, both numbers are 16-bit hexadecimal numbers, and the device manufacturer should provide them.

If you do not know the USB vendor ID and product ID, follow these steps to get them:

- a. Plug the device into the computer and allow the computer to recognize the new device. Cancel out of the Found New Hardware Wizard if it starts.
- b. Open the Device Manager from the Control Panel and find your device on the list, usually under Other Devices. It may show a yellow exclamation mark, indicating it is an unknown device.
- c. Double-click the device to open the properties.

- d. Select the **Details** tab and ensure that **Device Instance** Id shows in the attribute dropdown box. A string of characters is displayed similar to the following figure. The four characters to the right of **VID**\_ and **PID**\_ are your vendor ID and product ID, respectively.
- e. Write down the characters for your device, close the Device Manager, and unplug the device from the computer.

Alternately, you can contact your device vendor to obtain this information.

AQPad-6020E Properties 🔹 👔			
General Driver Details			
DAQPad-6020E			
Device Instance Id	~		
USB\VID_3923&PID_12C0\6&2B4AB4AF&0&4			

# Finding the Vendor ID and Product ID from the Device Manager

For the DAQPad-6020E, the vendor ID and product ID are 0x3923 and 0x12C0, respectively. The vendor ID and product ID are different for your device if it is not a National Instruments DAQPad-6020E.

Enter the vendor ID, product ID, manufacturer name, and model

name for your device in their respective fields.

Click **Next**. The **Output Files Generation** window is displayed as shown in the following figure.

🐝 NI-VISA Driver Wizard	
Output Files Generation	
Thank you for using the VISA Driver Development Wizard! This wizard will now gene which allow NI-VISA to control your device. Please specify the INF file name prefix a directory in which to write the files.	erate the files and the
If you are creating a distribution kit for your driver or application to be installed on o computers, you must copy the generated files into the appropriate directory. Instru doing this on a given operating system are included in the header of the generated	uther uctions for INF file.
Instrument Prefix (INF file name)	
DAQPad	
Directory in which to save the generated files:	
C:\VXIpnp\WinNT\DAQPad	Browse
Files to be generated:	
C:\VXIpnp\WinNT\DAQPad\DAQPad.inf C:\VXIpnp\WinNT\DAQPad\DAQPad_vista.inf	
< Back Next > Can	cel Help

#### **VISA DDW Output Files Generation Window**

3. The USB Instrument Prefix is a descriptor for identifying the files used for this device. Enter a USB instrument prefix, select the desired directory in which to place these files, and click **Next**. The next window displays installation options. The default selection is to install the setup information for the operating system and is usually the best option. Once you select an option, click **Finish** to exit the wizard. The .inf file is created in the directory you specified in the output file directory field in the previous window.

Note There are two .inf files. Choosing the Install the generated file(s) on this computer option in the next topic installs the correct file based on the OS. However, if you are installing the file manually, be sure to choose the correct file. For more information, refer to Using LabWindows/CVI to Install Your Device .inf Files.

Proceed to the next step, Installation Options.

## 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/2000, follow these steps:

- 1. Copy the .inf file to the inf folder. On Windows XP, this folder is usually at C:\Windows\inf. On Windows 2000, this folder is usually at C:\WINNT\inf. 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 Vista, follow these steps:

- 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.
- 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.

## Test Communication with VISA Interactive Control

To test communication with VISA Interactive Control (VISAIC), follow these steps:

 Open Measurement & Automation Explorer. Select Tools»Refresh to refresh the view. Your USB device should be listed under Devices and Interfaces, as shown in the following figure. 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.



#### **USB Device Shown in Measurement & Automation Explorer**

 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 Measurement & Automation Explorer. Select **Tools»NI-VISA»VISA Interactive Control**. A window similar to the following figure should open.



#### **VISA Interactive Control**

 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. The window shown in the following figure should open.

🗱 USB0::0x3923::0x12C0::00B50DAE::RAW (Session 0x00A99C58) 🛛 📃 🗖 🔀						
Template Basic I/O Interface I/O						
Enable Event Disable Event Discard Events Wait on Event Property Node (Set) Property Node (Get) Lock Unlock						
Attribute Name	Current Value					
Resource Name 🗾	USB0::0x3923::0x12C0::00B50DAE::RAW					
View All Settable Attributes	Return Status					
Query the value of the specified attribute.	<u>Execute</u>					

#### **VISA Interactive Control Open VISA Session**

When you open a VISA Session with VISAIC, the **Template** and **Property Node (Set)** tabs are automatically selected. To read a property, select the **Property Node (Get)** tab, choose the desired property, and click **Execute**. The **Current Value** indicator displays the current value of the property specified in **Attribute Name**. In the figure above, the **Resource Name** property was read.

For more information about using VISAIC, refer to the NI Developer Zone <u>VISA Interactive Control (VISAIC)</u> document. For a list of valid commands for your USB instrument, contact your instrument manufacturer.

# 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 488.2style 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.



**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 <u>USB Specification</u>.

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 <u>VI\_ATTR\_USB\_BULK\_IN\_PIPE</u> 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 <u>VI\_ATTR\_USB\_RECV\_INTR\_SIZE</u> and <u>VI\_ATTR\_USB\_RECV\_INTR\_DATA</u> attributes of the <u>VI\_EVENT\_USB\_INTR</u> event object.

## **USB on Linux and Mac**

### Linux

NI-VISA relies on a Linux kernel feature for its USB support. This feature is called *usbfs*, and was referred to as *usbdevfs* on older Linux kernels. For NI-VISA to support USB devices, this feature must be present and mounted (like a virtual file system). Most major Linux distributions such as Red Hat, SuSE, and Mandrake support this feature. You can use the mount command to display the currently mounted file systems 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 in /proc/bus/usb. If this is not the case, VISA cannot access the USB device (it cannot be found using <u>viFindRsrc</u>, and <u>viOpen</u> fails). On most systems, the root user has write access by default; 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. Use the hotplug package 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 installs 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 <u>http://linux-hotplug.sourceforge.net/</u>.

- Mount usbfs (formerly known as usbdevfs) with the option devmode=0666. This gives all users read and write access to all USB devices.
- Add write permissions to the file that represents the USB device, which is typically somewhere in a subdirectory in /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.

## **FireWire**®

NI-VISA supports programming FireWire devices only using INSTR access.

Introduction to Programming FireWire Devices in VISA

Configuring NI-VISA to Recognize an INSTR FireWire Device

FireWire Summary
# Introduction to Programming FireWire Devices in VISA

Users who are writing an application for a FireWire device can use NI-VISA to gain full access to all the device's registers. NI-VISA currently supports the FireWire interface on Windows and LabVIEW RT (Phar Lap ETS).

A FireWire resource is uniquely identified in NI-VISA by two characteristics: the FireWire vendor ID and the FireWire chip ID. Both IDs are extracted from the 64-bit GUID, which comes in two 32-bit halves. The vendor ID is the upper 24 bits of the upper half of the GUID. The chip ID is a combination of the lower 8 bits of the upper half of the GUID and the 32 bits of the lower half of the GUID. The canonical resource string you pass to viOpen() for a FireWire device is "FIREWIRE[board]::<vendor ID>::<chip ID>::INSTR". Currently, the only valid options for the board are 0 or None (that is, FIREWIRE0::<vendor ID>::<chip ID>::INSTR or FIREWIRE::<vendor ID>::<chip ID>::INSTR).

If you do not know the resource string, you can query the system with viFindRsrc() and use or display the resource(s) returned from that operation. You can create a query to search for devices of a particular attribute value. For example, if the FireWire vendor ID for National Instruments is 0xBE1394, you could find all National Instruments 1394 devices by calling viFindRsrc() with the expression "FIREWIRE? \*INSTR{VI\_ATTR\_FIREWIRE\_VENDOR\_ID==0xBE1394}".

# **Configuring NI-VISA to Recognize an INSTR FireWire Device**

Each FireWire 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** menu under **National Instruments»VISA**.

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

- Instrument Prefix—The VXI*plug&play* or IVI instrument driver prefix for the device.
- **Unit Specification ID**—The 24-bit value that identifies the specification to which the FireWire-based device adheres.
- Unit Software Version—The 24-bit value that identifies the Unit Specification version to which the FireWire-based device adheres.

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.

When you are done, the NI-VISA Driver Wizard generates a Windows Setup Information (.inf) file. Before a FireWire device is visible to NI-VISA, you must use the .inf file to update the Windows system registry. To manually install the .inf file, 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 before the wizard exits.

# **FireWire Summary**

NI-VISA provides a convenient means of accessing low-level functionality of FireWire devices. The alternative to using NI-VISA for FireWire 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.

# **Description of the API**

The following links take you to topics that describe the individual attributes, events, and operations. These are listed in alphabetical order within each access mechanism. Since a particular item can refer to more than one resource or interface type, each item is clearly marked with the resource and interface that support it.

Refer to <u>Resources</u> for a quick reference of how the attributes, events, and operations map to the available resources.

Attributes Events Operations VISA Access Mechanisms VISA Resource Types

# **VISA Access Mechanisms**

The following sections summarize the most important characteristics of attributes, events, and operations. Refer to <u>VISA Overview</u> for a more detailed description of this subject.

#### **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:

- <u>viGetAttribute</u> ()
- <u>viSetAttribute</u> ()

#### **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 <u>viEnableEvent</u> () operation with either the <u>viInstallHandler</u> () operation or the <u>viWaitOnEvent</u> () operation.

Events respond to attributes in the same manner that resources do. Once your application is done using a particular event received via <u>viWaitOnEvent</u> (), it should call <u>viClose</u> () 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:

- <u>viClose</u> ()
- <u>viGetAttribute</u> ()
- <u>viSetAttribute</u> ()
- viStatusDesc ()
- <u>viTerminate</u> ()
- <u>viLock</u> ()
- viUnlock ()
- <u>viEnableEvent</u> ()
- <u>viDisableEvent</u> ()
- <u>viDiscardEvents</u> ()
- <u>viWaitOnEvent</u> ()
- <u>viInstallHandler</u> ()
- <u>viUninstallHandler</u> ()

# **VISA Resource Types**

Currently, there are several VISA resource types—<u>INSTR Resource</u>, <u>MEMACC Resource</u>, <u>INTFC Resource</u>, <u>BACKPLANE Resource</u>, <u>SERVANT Resource</u>, <u>SOCKET Resource</u>, and <u>RAW Resource</u>. 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 deviceside 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 ([]).

Interface	Syntax
ENET-Serial INSTR	ASRL[0]::host address::serial port::INSTR
FireWire INSTR	FIREWIRE[ <i>board</i> ]::[vendorId]:: [chipId]::INSTR
GPIB INSTR	GPIB[board]::primary address[::secondary address][::INSTR]
GPIB INTFC	GPIB[board]::INTFC
GPIB-VXI BACKPLANE	GPIB-VXI[board][::VXI logical address]::BACKPLANE
GPIB-VXI INSTR	GPIB-VXI[board]::VXI logical address[::INSTR]
GPIB-VXI MEMACC	GPIB-VXI[board]::MEMACC
PXI 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 number[::FUNCfunction] [::INSTR]

PXI MEMACC	PXI[interface]::MEMACC
Remote NI-VISA	visa://host address[:server port]/remote resource
Serial INSTR	ASRLboard[::INSTR]
TCPIP INSTR	TCPIP[board]::host address[::LAN device name][::INSTR]
TCPIP SOCKET	TCPIP[board]::host address::port::SOCKET
USB INSTR	USB[board]::manufacturer ID::model code::serial number[::USB interface number] [::INSTR]
USB RAW	USB[board]::manufacturer ID::model code::serial number[::USB interface number]::RAW
VXI BACKPLANE	VXI[board][::VXI logical address]::BACKPLANE
VXI INSTR	VXI[board]::VXI logical address[::INSTR]
VXI MEMACC	VXI[board]::MEMACC
VXI 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 GPIB-VXI keyword for VXI resources via a GPIB-VXI controller. 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.

The following table shows the default value for optional string segments.

<b>Optional String Segments</b>	Default Value
board	0
GPIB secondary address	none
LAN device name	inst0
PXI bus	0
PXI function	0
USB interface number	lowest numbered relevant interface

The following table shows examples of address strings.

Address String	Description
ASRL::1.2.3.4::2::INSTR	A serial device attached to port 2 of the <u>ENET Serial</u> <u>controller</u> at address 1.2.3.4.
ASRL1::INSTR	A serial device attached to interface ASRL1.
FIREWIRE::0x123456:: 0x123456789A::INSTR	A FireWire device a with vendor ID of 0x123456 and a chip ID of 0x123456789A.
GPIB::1::0::INSTR	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB-VXI::9::INSTR	A VXI device at logical address 9 in a GPIB-VXI controlled system.

GPIB-VXI1::MEMACC	Board-level register access
	to GPIB-VXI interface
	number 1.

GPIB-

Mainframe resource for

VXI2::BACKPLANE	default chassis on GPIB-VXI interface 2.
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 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 inst0.
TCPIP0::1.2.3.4::999 ::SOCKET	Raw TCP/IP access to port 999 at the specified IP address.

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.
VXI::MEMACC	Board-level register access to the VXI interface.
VXI0::1::INSTR	A VXI device at logical

VXI0::SERVANT

address 1 in VXI interface VXI0.

Servant/device-side resource

for VXI interface 0.

# Attributes

These topics describe the VISA attributes. The attribute descriptions are listed in alphabetical order for easy reference.

Each attribute description contains a list below the title indicating the supported resource classes, such as GPIB, Serial, etc. The Attribute Information table lists the access privilege, the data type, range of values, and the default value.

# VI\_ATTR\_4882\_COMPLIANT

#### **Resource Classes**

GPIB-VXI INSTR, USB INSTR, VXI INSTR

## **Attribute Information**

Access Privilege	Data Type	Range	Default
Read Only	ViBoolean	VI_TRUE (1) VI_FALSE (0)	N/A
Global		() ()	

## Description

VI\_ATTR\_4882\_COMPLIANT specifies whether the device is 488.2 compliant.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_ASRL\_ALLOW\_TRANSMIT

## **Resource Classes**

Serial INSTR

## **Attribute Information**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_TRUE

## Description

If set to VI\_FALSE, it suspends transmission as if an XOFF character has been received. If set to VI\_TRUE, it resumes transmission as if an XON character has been received.

If XON/XOFF flow control (software handshaking) is not being used, it is invalid to set this attribute to VI\_FALSE.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_FLOW\_CNTRL

# VI\_ATTR\_ASRL\_AVAIL\_NUM

#### **Resource Classes**

Serial INSTR

## **Attribute Information**

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt32	0 to FFFFFFFh	N/A

#### Description

VI\_ATTR\_ASRL\_AVAIL\_NUM shows the number of bytes available in the low-level I/O receive buffer.

#### **Related Topics**

Controlling the Serial I/O Buffers INSTR Resource

# VI\_ATTR\_ASRL\_BAUD

#### **Resource Classes**

Serial INSTR

## **Attribute Information**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt32	0 to FFFFFFFh	9600

#### Description

VI\_ATTR\_ASRL\_BAUD is the baud rate of the interface. It is represented as an unsigned 32-bit integer so that any baud rate can be used, but it usually requires a commonly used rate such as 300, 1200, 2400, or 9600 baud.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_DATA\_BITS VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_PARITY VI\_ATTR\_ASRL\_STOP\_BITS

# VI\_ATTR\_ASRL\_BREAK\_LEN

#### **Resource Classes**

Serial INSTR

## **Attribute Information**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	1-500	250
This controls the duration (in milliseconds) of the break signal asserted when VI\_ATTR\_ASRL\_END\_OUT is set to VI\_ASRL\_END\_BREAK. If you want to control the assertion state and length of a break signal manually, use the VI\_ATTR\_ASRL\_BREAK\_STATE attribute instead.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BREAK\_STATE VI\_ATTR\_ASRL\_END\_OUT

# VI\_ATTR\_ASRL\_BREAK\_STATE

## **Resource Classes**

Access<br/>PrivilegeData<br/>TypeRangeDefaultRead/Write<br/>GlobalViInt16VI\_STATE\_ASSERTED(1)<br/>VI\_STATE\_UNASSERTED(1)<br/>(0)<br/>VI\_STATE\_UNKNOWN<br/>(-1)VI\_STATE\_UNASSERTED(1)<br/>(0)<br/>VI\_STATE\_UNKNOWN<br/>(-1)

If set to VI\_STATE\_ASSERTED, it suspends character transmission and places the transmission line in a break state until this attribute is reset to VI\_STATE\_UNASSERTED. This attribute lets you manually control the assertion state and length of a break signal. If you want VISA to send a break signal after each write operation automatically, use the VI\_ATTR\_ASRL\_BREAK\_LEN and VI\_ATTR\_ASRL\_END\_OUT attributes instead.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BREAK\_LEN VI\_ATTR\_ASRL\_END\_OUT VI\_ATTR\_ASRL\_ALLOW\_TRANSMIT

# VI\_ATTR\_ASRL\_CONNECTED

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read Onlv	ViBoolean	VI_TRUE (1) VI_FALSE (0)	N/A
Global			

VI\_ATTR\_ASRL\_CONNECTED indicates whether the port is properly connected to another port or device. This attribute is valid only with serial drivers developed by National Instruments and documented to support this feature with the corresponding National Instruments hardware.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_ASRL\_CTS\_STATE

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0) VI_STATE_UNKNOWN (- 1)	N/A

VI\_ATTR\_ASRL\_CTS\_STATE shows the current state of the Clear To Send (CTS) input signal.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_ASRL\_DTR\_STATE VI\_ATTR\_ASRL\_RI\_STATE VI\_ATTR\_ASRL\_RTS\_STATE

# VI\_ATTR\_ASRL\_DATA\_BITS

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt16	5 to 8	8

VI\_ATTR\_ASRL\_DATA\_BITS is the number of data bits contained in each frame (from 5 to 8). The data bits for each frame are located in the low-order bits of every byte stored in memory.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BAUD VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_PARITY VI\_ATTR\_ASRL\_STOP\_BITS

# VI\_ATTR\_ASRL\_DCD\_STATE

## **Resource Classes**

Туре	Range	Default
ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0)	N/A
	VI_STATE_UNKNOWN (- 1)	
	<b>Type</b> ViInt16	Type ViInt16 VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0) VI_STATE_UNKNOWN (- 1)

VI\_ATTR\_ASRL\_DCD\_STATE represents the current state of the Data Carrier Detect (DCD) input signal. The DCD signal is often used by modems to indicate the detection of a carrier (remote modem) on the telephone line. The DCD signal is also known as Receive Line Signal Detect (RLSD). This attribute is Read Only except when the VI\_ATTR\_ASRL\_WIRE\_MODE attribute is set to VI\_ASRL\_WIRE\_232\_DCE, or VI\_ASRL\_WIRE\_232\_AUTO with the hardware currently in the DCE state.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_ASRL\_DTR\_STATE VI\_ATTR\_ASRL\_RI\_STATE VI\_ATTR\_ASRL\_RTS\_STATE

## VI\_ATTR\_ASRL\_DISCARD\_NULL

Note This attribute 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.

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

If set to VI\_TRUE, NUL characters are discarded. Otherwise, they are treated as normal data characters. For binary transfers, set this attribute to VI\_FALSE.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_ASRL\_DSR\_STATE

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0) VI_STATE_UNKNOWN (- 1)	N/A

VI\_ATTR\_ASRL\_DSR\_STATE shows the current state of the Data Set Ready (DSR) input signal.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_ASRL\_DTR\_STATE VI\_ATTR\_ASRL\_RI\_STATE VI\_ATTR\_ASRL\_RTS\_STATE

# VI\_ATTR\_ASRL\_DTR\_STATE

## **Resource Classes**

Туре	Range	Default
ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0)	N/A
	VI_STATE_UNKNOWN (- 1)	
	<b>Type</b> ViInt16	Type ViInt16 VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0) VI_STATE_UNKNOWN (- 1)

VI\_ATTR\_ASRL\_DTR\_STATE shows the current state of the Data Terminal Ready (DTR) input signal.

When the VI\_ATTR\_ASRL\_FLOW\_CNTRL attribute is set to VI\_ASRL\_FLOW\_DTR\_DSR, this attribute is Read Only. Querying the value will return VI\_STATE\_UNKNOWN.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_ASRL\_RI\_STATE VI\_ATTR\_ASRL\_RTS\_STATE

# VI\_ATTR\_ASRL\_END\_IN

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_ASRL_END_NONE (0) VI_ASRL_END_LAST_BIT (1) VI_ASRL_END_TERMCHAR (2)	VI_ASRL_END_TERM

VI\_ATTR\_ASRL\_END\_IN indicates the method used to terminate read operations.

- If it is set to VI\_ASRL\_END\_NONE, the read will not terminate until all of the requested data is received (or an error occurs).
- If it is set to VI\_ASRL\_END\_LAST\_BIT, the read will terminate as soon as a character arrives with its last bit set. For example, if VI\_ATTR\_ASRL\_DATA\_BITS is set to 8, the read will terminate when a character arrives with the 8th bit set.
- If it is set to VI\_ASRL\_END\_TERMCHAR, the read will terminate as soon as the character in VI\_ATTR\_TERMCHAR is received. In this case, VI\_ATTR\_TERMCHAR\_EN is ignored.

Because the default value of VI\_ATTR\_TERMCHAR is 0Ah (linefeed), read operations on serial ports will stop reading whenever a linefeed is encountered. To change this behavior, you must change the value of one of these attributes—VI\_ATTR\_ASRL\_END\_IN or VI\_ATTR\_TERMCHAR.

### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_END\_OUT VI\_ATTR\_TERMCHAR

# VI\_ATTR\_ASRL\_END\_OUT

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_ASRL_END_NONE (0) VI_ASRL_END_LAST_BIT (1) VI_ASRL_END_TERMCHAR (2) VI_ASRL_END_BREAK (3)	VI_ASRL_END_NONI

VI\_ATTR\_ASRL\_END\_OUT indicates the method used to terminate write operations.

- If it is set to VI\_ASRL\_END\_NONE, the write will transmit the exact contents of the user buffer, without modifying it and without appending anything to the data being written.
- If it is set to VI\_ASRL\_END\_LAST\_BIT, and VI\_ATTR\_SEND\_END\_EN is set to VI\_TRUE, the write will send all but the last character with the highest bit clear, then transmit the last character with the highest bit set. For example, if VI\_ATTR\_ASRL\_DATA\_BITS is set to 8, the write will clear the eighth bit for all but the last character, then transmit the last character with the eighth bit set. If VI\_ATTR\_SEND\_END\_EN is set to VI\_FALSE, the write will send all the characters with the highest bit clear.
- If it is set to VI\_ASRL\_END\_TERMCHAR, and VI\_ATTR\_SEND\_END\_EN is set to VI\_TRUE, the write will send the character in VI\_ATTR\_TERMCHAR after the data being transmitted. If VI\_ATTR\_SEND\_END\_EN is set to VI\_FALSE, the write will transmit the exact contents of the user buffer, without modifying it and without appending anything to the data being written.
- If it is set to VI\_ASRL\_END\_BREAK, and
- VI\_ATTR\_SEND\_END\_EN is set to VI\_TRUE, the write will transmit a break after all the characters for the write have been sent. If VI\_ATTR\_SEND\_END\_EN is set to VI\_FALSE, the write will transmit the exact contents of the user buffer, without modifying it and without appending anything to the data being written.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_END\_IN VI\_ATTR\_TERMCHAR

# VI\_ATTR\_ASRL\_FLOW\_CNTRL

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt16	VI_ASRL_FLOW_NONE (0) VI_ASRL_FLOW_XON_XOFF	VI_ASRL_FLOW_NC
		(1)	
		VI_ASRL_FLOW_RTS_CTS	
		(2)	
		VI_ASRL_FLOW_DTR_DSR	
		(4)	

VI\_ATTR\_ASRL\_FLOW\_CNTRL indicates the type of flow control used by the transfer mechanism.

- If this attribute is set to VI\_ASRL\_FLOW\_NONE, the transfer mechanism does not use flow control, and buffers on both sides of the connection are assumed to be large enough to hold all data transferred.
- If this attribute is set to VI\_ASRL\_FLOW\_XON\_XOFF, the transfer mechanism uses the XON and XOFF characters to perform flow control. The transfer mechanism controls input flow by sending XOFF when the low-level I/O receive buffer is nearly full, and it controls the output flow by suspending transmission when XOFF is received.
- If this attribute is set to VI\_ASRL\_FLOW\_RTS\_CTS, the transfer mechanism uses the RTS output signal and the CTS input signal to perform flow control. The transfer mechanism controls input flow by unasserting the RTS signal when the low-level I/O receive buffer is nearly full, and it controls output flow by suspending the transmission when the CTS signal is unasserted.
- If this attribute is set to VI\_ASRL\_FLOW\_DTR\_DSR, the transfer mechanism uses the DTR output signal and the DSR input signal to perform flow control. The transfer mechanism controls input flow by unasserting the DTR signal when the low-level I/O receive buffer is nearly full, and it controls output flow by suspending the transmission when the DSR signal is unasserted.

This attribute can specify multiple flow control mechanisms by bit-ORing multiple values together. However, certain combinations may not be supported by all serial ports and/or operating systems.

### **Related Topics**

Controlling the Serial I/O Buffers INSTR Resource VI\_ATTR\_ASRL\_BAUD VI\_ATTR\_ASRL\_DATA\_BITS VI\_ATTR\_ASRL\_PARITY VI\_ATTR\_ASRL\_STOP\_BITS VI\_ATTR\_ASRL\_XOFF\_CHAR VI\_ATTR\_ASRL\_XON\_CHAR
# VI\_ATTR\_ASRL\_PARITY

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt16	VI_ASRL_PAR_NONE (0) VI_ASRL_PAR_ODD (1) VI_ASRL_PAR_EVEN (2) VI_ASRL_PAR_MARK (3) VI_ASRL_PAR_SPACE	VI_ASRL_PAR_NONE
		(4)	

VI\_ATTR\_ASRL\_PARITY is the parity used with every frame transmitted and received.

- VI\_ASRL\_PAR\_MARK means that the parity bit exists and is always 1.
- VI\_ASRL\_PAR\_SPACE means that the parity bit exists and is always 0.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BAUD VI\_ATTR\_ASRL\_DATA\_BITS VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_STOP\_BITS

# VI\_ATTR\_ASRL\_REPLACE\_CHAR

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt8	0 to FFh	0

VI\_ATTR\_ASRL\_REPLACE\_CHAR specifies the character to be used to replace incoming characters that arrive with errors (such as parity error).

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_PARITY

# VI\_ATTR\_ASRL\_RI\_STATE

### **Resource Classes**

Туре	Range	Default
ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0)	N/A
	VI_STATE_UNKNOWN (- 1)	
	<b>Type</b> ViInt16	Type ViInt16 VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0) VI_STATE_UNKNOWN (- 1)

VI\_ATTR\_ASRL\_RI\_STATE represents the current state of the Ring Indicator (RI) input signal. The RI signal is often used by modems to indicate that the telephone line is ringing. This attribute is Read Only except when the VI\_ATTR\_ASRL\_WIRE\_MODE attribute is set to VI\_ASRL\_WIRE\_232\_DCE, or VI\_ASRL\_WIRE\_232\_AUTO with the hardware currently in the DCE state.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_ASRL\_DTR\_STATE VI\_ATTR\_ASRL\_RTS\_STATE

# VI\_ATTR\_ASRL\_RTS\_STATE

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViInt16	VI_STATE_ASSERTED (1) VI_STATE_UNASSERTED (0)	N/A
		VI_STATE_UNKNOWN (- 1)	

VI\_ATTR\_ASRL\_RTS\_STATE is used to manually assert or unassert the Request To Send (RTS) output signal.

When the VI\_ATTR\_ASRL\_FLOW\_CNTRL attribute is set to VI\_ASRL\_FLOW\_RTS\_CTS, this attribute is Read Only. Querying the value will return VI\_STATE\_UNKNOWN.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_ASRL\_DTR\_STATE VI\_ATTR\_ASRL\_RI\_STATE

# VI\_ATTR\_ASRL\_STOP\_BITS

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt16	VI_ASRL_STOP_ONE (10)	VI_ASRL_STOP_ONE
		VI_ASRL_STOP_ONE5	
		(15)	
		VI_ASRL_STOP_TWO	
		(20)	

VI\_ATTR\_ASRL\_STOP\_BITS is the number of stop bits used to indicate the end of a frame. The value VI\_ASRL\_STOP\_ONE5 indicates one-and-one-half (1.5) stop bits.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BAUD VI\_ATTR\_ASRL\_DATA\_BITS VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_PARITY

# VI\_ATTR\_ASRL\_WIRE\_MODE

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViInt16	VI_ASRL_WIRE_485_4 (0) VI_ASRL_WIRE_485_2_DTR_ECHO (1)	N/A
		VI_ASRL_WIRE_485_2_DTR_CTRL (2) VI_ASRL_WIRE_485_2_AUTO (3) VI_ASRL_WIRE_232_DTE (128) VI_ASRL_WIRE_232_DCE (129)	
		VI_ASRL_WIRE_232_AUTO (130)VI_STATE_UNKNOWN (-1)	

VI\_ATTR\_ASRL\_WIRE\_MODE represents the current wire/transceiver mode.

For RS-485 hardware, this attribute is valid only with the RS-485 serial driver developed by National Instruments.

For RS-232 hardware, the values RS232/DCE and RS232/AUTO are valid only with RS-232 serial drivers developed by National Instruments and documented to support this feature with the corresponding National Instruments hardware. When this feature is not supported, RS232/DTE is the only valid value.

RS-232 settings:

- VI\_ASRL\_WIRE\_232\_DTE uses DTE mode.
- VI\_ASRL\_WIRE\_232\_DCE uses DCE mode.
- VI\_ASRL\_WIRE\_232\_AUTO automatically detects which mode to use.

RS-485 settings:

- VI\_ASRL\_WIRE\_485\_4 uses 4-wire mode.
- VI\_ASRL\_WIRE\_485\_2\_DTR\_ECHO uses 2-wire DTR mode controlled with echo.
- VI\_ASRL\_WIRE\_485\_2\_DTR\_CTRL uses 2-wire DTR mode controlled without echo.
- VI\_ASRL\_WIRE\_485\_2\_AUTO uses 2-wire auto mode controlled with TXRDY.
- Note This attribute is valid only on the platforms on which National Instruments supports its RS-232 or RS-485 products.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_ASRL\_XOFF\_CHAR

## **Resource Classes**

Access Privilege	Data Type	Range	Default	
Read/Write Local	ViUInt8	0 to FFh	<control-s> (13h)</control-s>	

VI\_ATTR\_ASRL\_XOFF\_CHAR specifies the value of the XOFF character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_XON\_CHAR

# VI\_ATTR\_ASRL\_XON\_CHAR

## **Resource Classes**

Access Privilege	Data Type	Range	Default	
Read/Write Local	ViUInt8	0 to FFh	<control-q> (11h)</control-q>	

VI\_ATTR\_ASRL\_XON\_CHAR specifies the value of the XON character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_FLOW\_CNTRL VI\_ATTR\_ASRL\_XOFF\_CHAR

# VI\_ATTR\_BUFFER

### **Resource Classes**

VI\_EVENT\_IO\_COMPLETION

Access Data Privilege Type		Range	Default	
Read Only	ViBuf	N/A	N/A	

VI\_ATTR\_BUFFER contains the address of a buffer that was used in an asynchronous operation.

#### **Related Topics**

VI\_ATTR\_JOB\_ID

VI\_ATTR\_RET\_COUNT/VI\_ATTR\_RET\_COUNT\_32/VI\_ATTR\_RET\_COU

VI\_EVENT\_IO\_COMPLETION

# VI\_ATTR\_CMDR\_LA

#### **Resource Classes**

GPIB-VXI INSTR, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read	ViInt16	0 to 255 VL UNKNOWN LA	N/A
Global		(-1)	

VI\_ATTR\_CMDR\_LA is the unique logical address of the commander of the VXI device used by the given session.

#### **Related Topics**

INSTR Resource SERVANT Resource

# VI\_ATTR\_DEST\_ACCESS\_PRIV

#### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	ess Data Range lege Type		Default	
Read/Write Local	ViUInt16	VI_DATA_PRIV (0)	VI_DATA_PRIV	
		VI_DATA_NPRIV		
		(1)		
		VI_PROG_PRIV		
		(2)		
		VI_PROG_NPRIV		
		(3)		
		VI_BLCK_PRIV		
		(4)		
		VI_BLCK_NPRIV		
		(5)		
		VI_D64_PRIV (6)		
		VI_D64_NPRIV		
		(7)		

VI\_ATTR\_DEST\_ACCESS\_PRIV specifies the address modifier to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.



**Note** The values VI\_D64\_PRIV (6) and VI\_D64\_NPRIV (7) apply to only the block move operations. If you set this attribute to one of these values and then call one of the viOutXX() operations, the operation returns VI\_ERROR\_INV\_SETUP.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_BYTE\_ORDER VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_SRC\_ACCESS\_PRIV VI\_ATTR\_WIN\_ACCESS\_PRIV

# VI\_ATTR\_DEST\_BYTE\_ORDER

#### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range		C	Default	
Read/Write Local	ViUInt16	VI_BIG_ VI_LITT	_ENDIAN ( `LE_ENDIA (1)	(0) V AN	/I_BIG_	ENDIAN_

VI\_ATTR\_DEST\_BYTE\_ORDER specifies the byte order to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_ACCESS\_PRIV VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_SRC\_BYTE\_ORDER VI\_ATTR\_WIN\_BYTE\_ORDER
# VI\_ATTR\_DEST\_INCREMENT

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt32	0 to 1	1

VI\_ATTR\_DEST\_INCREMENT is used in the viMoveOutXX() operations to specify by how many elements the destination offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOutXX() operations move into consecutive elements. If this attribute is set to 0, the viMoveOutXX() operations will always write to the same element, essentially treating the destination as a FIFO register.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_ACCESS\_PRIV VI\_ATTR\_DEST\_BYTE\_ORDER VI\_ATTR\_SRC\_INCREMENT

# VI\_ATTR\_DEV\_STATUS\_BYTE

## **Resource Classes**

GPIB INTFC, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Global	ViUInt8	0 to FFh	N/A

This attribute specifies the 488-style status byte of the local controller or device associated with this session.

If this attribute is written and bit 6 (40h) is set, this device or controller will assert a service request (SRQ) if it is defined for this interface.

#### **Related Topics**

INTFC Resource SERVANT Resource

# VI\_ATTR\_DMA\_ALLOW\_EN

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, Serial INSTR, TCPIP INSTR, VXI INSTR, VXI MEMACC, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE(1) VI_FALSE(0)	N/A

This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

#### **Related Topics**

INSTR Resource INTFC Resource MEMACC Resource SERVANT Resource

# VI\_ATTR\_EVENT\_TYPE

## **Resource Classes**

All event object types

Access Privilege	Data Type	Range	Default
Read Only	ViEventType	0h to FFFFFFFh	N/A

VI\_ATTR\_EVENT\_TYPE is the unique logical identifier for the event type of the specified event.

### **Related Topics**

**Events** 

# VI\_ATTR\_FDC\_CHNL

### **Resource Classes**

**VXI INSTR** 

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	0 to 7	N/A

VI\_ATTR\_FDC\_CHNL determines which Fast Data Channel (FDC) will be used to transfer the buffer.

#### **Related Topics**

INSTR Resource VI\_ATTR\_FDC\_MODE VI\_ATTR\_FDC\_USE\_PAIR

# VI\_ATTR\_FDC\_MODE

### **Resource Classes**

**VXI INSTR** 

Access Privilege	Data Type	Range		Default	
Read/Write Local	ViUInt16	VI_FDC_	_NORMAL (1)	VI_FDC_	NORMAL
		VI_FDC	_STREAM		
		(	(2)		

VI\_ATTR\_FDC\_MODE specifies which Fast Data Channel (FDC) mode to use (either normal or stream mode).

#### **Related Topics**

INSTR Resource VI\_ATTR\_FDC\_CHNL VI\_ATTR\_FDC\_USE\_PAIR

# VI\_ATTR\_FDC\_USE\_PAIR

**Resource Classes** 

**VXI INSTR** 

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

Setting VI\_ATTR\_FDC\_USE\_PAIR to VI\_TRUE specifies to use a channel pair for transferring data. Otherwise, only one channel will be used.

#### **Related Topics**

INSTR Resource VI\_ATTR\_FDC\_CHNL VI\_ATTR\_FDC\_MODE

# VI\_ATTR\_FILE\_APPEND\_EN

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

This attribute specifies whether viReadToFile() will overwrite (truncate) or append when opening a file.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viReadToFile

# VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET

## **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	0 to FFFFh	0xFFFF

This property specifies the upper 16 bits of the 48-bit destination address for a FireWire device. This attribute is used in correlation with the viOut8/16/32, viMove, and viMoveAsync methods.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET

# VI\_ATTR\_FIREWIRE\_LOWER\_32\_CHIP\_ID

#### **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read- Only Global	ViUInt32	0 to FFFFFFFh	N/A

This property specifies the lower chip ID for a FireWire device.

#### **Related Topics**

INSTR Resource

VI\_ATTR\_FIREWIRE\_UPPER\_8\_CHIP\_ID

VI\_ATTR\_FIREWIRE\_VENDOR\_ID

# VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET

#### **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	0 to FFFFh	0xFFFF

This property specifies the upper 16 bits of the 48-bit source address for a FireWire device. This attribute is used in correlation with the viIn8/16/32, viMove, and viMoveAsync methods.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET

# VI\_ATTR\_FIREWIRE\_UPPER\_8\_CHIP\_ID

### **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read- Only Global	ViUInt8	0 to FFh	N/A

This property specifies the upper chip ID for a FireWire device.

#### **Related Topics**

INSTR Resource

VI\_ATTR\_FIREWIRE\_LOWER\_32\_CHIP\_ID

VI\_ATTR\_FIREWIRE\_VENDOR\_ID
# VI\_ATTR\_FIREWIRE\_VENDOR\_ID

## **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read- Only Global	ViUInt32	0 to FFFFFFh	N/A

This property specifies the vendor ID for a FireWire device.

### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_FIREWIRE\_LOWER\_32\_CHIP\_ID

VI\_ATTR\_FIREWIRE\_UPPER\_8\_CHIP\_ID

# VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET

## **Resource Classes**

FireWire INSTR

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	0 to FFFFh	0xFFFF

This property specifies the upper 16 bits of the 48-bit address for a FireWire device when a window is mapped. This attribute is used in correlation with the viPeek8/16/32, viPoke8/16/32, and viMapAddress methods.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET

# VI\_ATTR\_GPIB\_ADDR\_STATE

### **Resource Classes**

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_GPIB_UNADDRESSED(0)	N/A
Only		VI_GPIB_TALKER(1)	
Global		VI_GPIB_LISTENER(2)	

This attribute shows whether the specified GPIB interface is currently addressed to talk or listen, or is not addressed.

#### **Related Topics**

INTFC Resource SERVANT Resource

# VI\_ATTR\_GPIB\_ATN\_STATE

# **Resource Classes**

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_STATE_ASSERTED(1)	N/A
Only		VI_STATE_UNASSERTED(0)	
Global		VI_STATE_UNKNOWN(-1)	

This attribute shows the current state of the GPIB ATN (ATtentioN) interface line.

#### **Related Topics**

INTFC Resource VI\_ATTR\_GPIB\_NDAC\_STATE VI\_ATTR\_GPIB\_REN\_STATE VI\_ATTR\_GPIB\_SRQ\_STATE viGpibControlATN

# VI\_ATTR\_GPIB\_CIC\_STATE

**Resource Classes** 

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViBoolean	VI_TRUE(1) VI_FALSE(0)	N/A

This attribute shows whether the specified GPIB interface is currently CIC (Controller In Charge).

#### **Related Topics**

INTFC Resource

VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE

# VI\_ATTR\_GPIB\_HS488\_CBL\_LEN

**Resource Classes** 

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read/Write Global	ViInt16	VI_GPIB_HS488_NIMPL(-1) VI_GPIB_HS488_DISABLED(0) 1–15	N/A

This attribute specifies the total number of meters of GPIB cable used in the specified GPIB interface.

#### **Related Topics**

INTFC Resource VI\_ATTR\_IO\_PROT

# VI\_ATTR\_GPIB\_NDAC\_STATE

# **Resource Classes**

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_STATE_ASSERTED(1)	N/A
Only		VI_STATE_UNASSERTED(0)	
Global		VI_STATE_UNKNOWN(-1)	

This attribute shows the current state of the GPIB NDAC (Not Data ACcepted) interface line.

#### **Related Topics**

INTFC Resource VI\_ATTR\_GPIB\_ATN\_STATE VI\_ATTR\_GPIB\_REN\_STATE VI\_ATTR\_GPIB\_SRQ\_STATE

# VI\_ATTR\_GPIB\_PRIMARY\_ADDR

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI MEMACC, GPIB-VXI BACKPLANE

Access Privilege	Data Type	Range	Default
INSTR, MEMACC, BACKPLANE: Read Only Global	ViUInt16	0 to 30	N/A
INTFC: Read/Write Global			

VI\_ATTR\_GPIB\_PRIMARY\_ADDR specifies the primary address of the GPIB device used by the given session. For the GPIB INTFC Resource, this attribute is Read-Write.

#### **Related Topics**

BACKPLANE Resource

**INSTR Resource** 

INTFC Resource

MEMACC Resource

VI\_ATTR\_GPIB\_SECONDARY\_ADDR

# VI\_ATTR\_GPIB\_READDR\_EN

### **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_TRUE

VI\_ATTR\_GPIB\_READDR\_EN specifies whether to use repeat addressing before each read or write operation.

#### **Related Topics**

INSTR Resource VI\_ATTR\_GPIB\_UNADDR\_EN

# VI\_ATTR\_GPIB\_RECV\_CIC\_STATE

### **Resource Classes**

VI\_EVENT\_GPIB\_CIC

Access Privilege	Data Type	Range	Default
Read- Only	ViBoolean	VI_TRUE (1) VI_FALSE (0)	N/A

This attribute specifies whether the local controller has gained or lost CIC status.

### **Related Topics**

INTFC Resource VI\_ATTR\_GPIB\_ATN\_STATE VI\_EVENT\_GPIB\_CIC

# VI\_ATTR\_GPIB\_REN\_STATE

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_STATE_ASSERTED(1)	N/A
Only		VI_STATE_UNASSERTED(0)	
Global		VI_STATE_UNKNOWN(-1)	

VI\_ATTR\_GPIB\_REN\_STATE returns the current state of the GPIB REN (Remote ENable) interface line.

#### **Related Topics**

INSTR Resource INTFC Resource VI\_ATTR\_FILE\_APPEND\_EN VI\_ATTR\_GPIB\_ATN\_STATE VI\_ATTR\_GPIB\_NDAC\_STATE VI\_ATTR\_GPIB\_SRQ\_STATE viGpibControlREN

# VI\_ATTR\_GPIB\_SECONDARY\_ADDR

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI MEMACC, GPIB-VXI BACKPLANE

Access Privilege	Data Type	Range	Default
INSTR, MEMACC, BACKPLANE: Read Only Global	ViUInt16	0 to 30, VI_NO_SEC_ADDR (FFFFh)	N/A
INTFC: Read/Write Global			

VI\_ATTR\_GPIB\_SECONDARY\_ADDR specifies the secondary address of the GPIB device used by the given session. For the GPIB INTFC Resource, this attribute is Read-Write.

#### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource MEMACC Resource VI\_ATTR\_GPIB\_PRIMARY\_ADDR
# VI\_ATTR\_GPIB\_SRQ\_STATE

**Resource Classes** 

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_STATE_ASSERTED(1)	N/A
Only		VI_STATE_UNASSERTED(0)	
Global		VI_STATE_UNKNOWN(-1)	

This attribute shows the current state of the GPIB SRQ (Service ReQuest) interface line.

### **Related Topics**

INTFC Resource VI\_ATTR\_GPIB\_ATN\_STATE VI\_ATTR\_GPIB\_NDAC\_STATE VI\_ATTR\_GPIB\_REN\_STATE

# VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE

## **Resource Classes**

**GPIB INTFC** 

Access Privilege	Data Type	Range	Default
Read/Write Global	ViBoolean	VI_TRUE(1) VI_FALSE(0)	N/A

This attribute shows whether the specified GPIB interface is currently the system controller. In some implementations, this attribute may be modified only through a configuration utility. On these systems this attribute is read-only (RO).

#### **Related Topics**

INTFC Resource VI\_ATTR\_GPIB\_CIC\_STATE

# VI\_ATTR\_GPIB\_UNADDR\_EN

## **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

VI\_ATTR\_GPIB\_UNADDR\_EN specifies whether to unaddress the device (UNT and UNL) after each read or write operation.

#### **Related Topics**

INSTR Resource VI\_ATTR\_GPIB\_READDR\_EN

# VI\_ATTR\_IMMEDIATE\_SERV

### **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only	ViBoolean	VI_TRUE (1) VI_FALSE (0)	N/A
Global			

VI\_ATTR\_IMMEDIATE\_SERV specifies whether the device associated with this session is an immediate servant of the controller running VISA.

### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_INTF\_INST\_NAME

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

VI\_ATTR\_INTF\_INST\_NAME specifies human-readable text that describes the given interface.



**Note** The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

#### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource MEMACC Resource SOCKET Resource VI\_ATTR\_INTF\_NUM VI\_ATTR\_INTF\_TYPE

# VI\_ATTR\_INTF\_NUM

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0h to FFFFh	0

VI\_ATTR\_INTF\_NUM specifies the board number for the given interface.

Related Topics BACKPLANE Resource INSTR Resource INTFC Resource MEMACC Resource SOCKET Resource VI\_ATTR\_INTF\_TYPE

# VI\_ATTR\_INTF\_PARENT\_NUM

#### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, GPIB-VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0h to FFFFh	0

VI\_ATTR\_INTF\_PARENT\_NUM specifies the board number of the GPIB board to which the GPIB-VXI is attached.

#### **Related Topics**

BACKPLANE Resource INTFC Resource MEMACC Resource VI\_ATTR\_INTF\_NUM VI\_ATTR\_INTF\_TYPE

# VI\_ATTR\_INTF\_TYPE

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read	ViUInt16	VI_INTF_GPIB (1)	N/A
Only		VI_INTF_VXI (2)	
Global		VI_INTF_GPIB_VXI	
		(3)	
		VI_INTF_ASRL (4)	
		VI_INTF_PXI (5)	
		VI_INTF_TCPIP (6)	
		VI_INTF_USB (7)	

VI\_ATTR\_INTF\_TYPE specifies the interface type of the given session.

### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource MEMACC Resource SOCKET Resource VI\_ATTR\_INTF\_NUM

# VI\_ATTR\_INTR\_STATUS\_ID

**Resource Classes** 

VI\_EVENT\_VXI\_VME\_INTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt32	0 to FFFFFFFh	N/A

VI\_ATTR\_INTR\_STATUS\_ID specifies the 32-bit status/ID retrieved during the IACK cycle.

#### **Related Topics**

INSTR Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_RECV\_INTR\_LEVEL VI\_EVENT\_VXI\_VME\_INTR

# VI\_ATTR\_IO\_PROT

#### **Resource Classes**

GPIB INTFC, GPIB INSTR, GPIB-VXI INSTR, Serial INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access	Data	Dango
Privilege	Туре	Range

Read/Write ViUInt16 Local

**GPIB**: VI\_PROT\_NORMAL (1) VI\_PROT\_HS488 (3)

Default

VI\_PROT\_NORMAL

VI\_PROT\_NORMAL

VXI VI\_PROT\_NORMAL (1) VI\_PROT\_FDC (2)

VI PROT NORMAL

**GPIB-VXI**: VI\_PROT\_NORMAL (1)

Serial, TCPIP, USB RAW: VI\_PROT\_NORMAL (1)

VI PROT NORMAL

VI\_PROT\_4882\_STRS (4)

VI\_PROT\_NORMAL

**USB INSTR:** VI\_PROT\_NORMAL (1) VI\_PROT\_USBTMC\_VENDOR (5)

VI\_ATTR\_IO\_PROT specifies which protocol to use. In VXI, you can choose normal word serial or fast data channel (FDC). In GPIB, you can choose normal or high-speed (HS-488) transfers. In serial, TCPIP, or USB RAW, you can choose normal transfers or 488.2-defined strings. In USB INSTR, you can choose normal or vendor-specific transfers.

In previous versions of VISA, VI\_PROT\_NORMAL was known as VI\_NORMAL, VI\_PROT\_FDC was known as VI\_FDC, VI\_PROT\_HS488 was known as VI\_HS488, and VI\_PROT\_4882\_STRS was known as VI\_ASRL488.

#### **Related Topics**

INSTR Resource INTFC Resource RAW Resource SERVANT Resource SOCKET Resource VI\_ATTR\_FDC\_CHNL VI\_ATTR\_FDC\_MODE VI\_ATTR\_FDC\_USE\_PAIR VI\_ATTR\_FDB\_HS488\_CBL\_LEN

# VI\_ATTR\_JOB\_ID

**Resource Classes** 

VI\_EVENT\_IO\_COMPLETION

Access Privilege	Data Type	Range	Default
Read Only	ViJobId	N/A	N/A

VI\_ATTR\_JOB\_ID contains the job ID of the asynchronous operation that has completed.

#### **Related Topics**

**Resources** 

SERVANT Resource

VI\_ATTR\_BUFFER

VI\_ATTR\_RET\_COUNT/VI\_ATTR\_RET\_COUNT\_32/VI\_ATTR\_RET\_COU

VI\_ATTR\_STATUS

VI\_EVENT\_IO\_COMPLETION

# VI\_ATTR\_MAINfRAME\_LA

### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI BACKPLANE, VXI INSTR, VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to 255 VI_UNKNOWN_LA (-1)	N/A

VI\_ATTR\_MA.infRAME\_LA specifies the lowest logical address in the mainframe. If the logical address is not known, VI\_UNKNOWN\_LA is returned.

#### **Related Topics**

**BACKPLANE Resource** 

**INSTR Resource**
## VI\_ATTR\_MANF\_ID

#### **Resource Classes**

GPIB-VXI INSTR, PXI INSTR, USB INSTR, USB RAW, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0h to FFFFh	N/A

VI\_ATTR\_MANF\_ID is the manufacturer identification number of the device.

For VXI resources, this refers to the VXI Manufacturer ID.

For PXI INSTR resources, if the subsystem PCI Vendor ID is nonzero, this refers to the subsystem Vendor ID. Otherwise, this refers to the Vendor ID.

For USB resources, this refers to the Vendor ID (VID).

### **Related Topics**

INSTR Resource VI\_ATTR\_MANF\_NAME VI\_ATTR\_MODEL\_CODE

## VI\_ATTR\_MANF\_NAME

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, PXI INSTR, PXI BACKPLANE, USB INSTR, USB RAW, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

This string attribute is the manufacturer name.

Note The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

#### **Related Topics**

INSTR Resource VI\_ATTR\_MANF\_ID VI\_ATTR\_MODEL\_NAME

# VI\_ATTR\_MAX\_QUEUE\_LENGTH

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read/Write	ViUInt32	1h to	50
Local		FFFFFFFh	

VI\_ATTR\_MAX\_QUEUE\_LENGTH specifies the maximum number of events that can be queued at any time on the given session. Events that occur after the queue has become full will be discarded.

VI\_ATTR\_MAX\_QUEUE\_LENGTH is a Read/Write attribute until the first time viEnableEvent() is called on a session. Thereafter, this attribute is Read Only.

#### **Related Topics**

Operations viEnableEvent VISA Resource Template viWaitOnEvent

## VI\_ATTR\_MEM\_BASE/VI\_ATTR\_MEM\_BASE\_32/\

#### **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

Access Privilege Data Type

Read

Only

Global

Range

#### Defa

VI\_ATTR\_MEM\_BASE: N// Oh to FFFFFFFh for 32-bit applications

VI\_ATTR\_MEM\_BASE\_32: ViUInt32

VI\_ATTR\_MEM\_BASE:

ViBusAddress

VI\_ATTR\_MEM\_BASE\_64: ViUInt64

0h to FFFFFFFFFFFFFFF for 64-bit applications

VI\_ATTR\_MEM\_BASE\_32: Oh to FFFFFFh

VI\_ATTR\_MEM\_BASE\_64: Oh to FFFFFFFFFFFFFFF

VI\_ATTR\_MEM\_BASE, VI\_ATTR\_MEM\_BASE\_32, and VI\_ATTR\_MEM\_BASE\_64 specify the base address of the device in VXIbus memory address space. This base address is applicable to A24 or A32 address space. If the value of VI\_ATTR\_MEM\_SPACE is VI\_A16\_SPACE, the value of this attribute is meaningless for the given VXI device.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_MEM\_SIZE/VI\_ATTR\_MEM\_SIZE\_32/VI\_ATTR\_MEM\_SIZE\_6<sup>,</sup> VI\_ATTR\_MEM\_SPACE

## VI\_ATTR\_MEM\_SIZE/VI\_ATTR\_MEM\_SIZE\_32/VI\_

#### **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

Access Privilege Data Type

Read VI\_ATTR\_MEM\_SIZE: Only ViBusSize Global

> VI\_ATTR\_MEM\_SIZE\_32: ViUInt32

> VI\_ATTR\_MEM\_SIZE\_64: ViUInt64

#### Range

#### Default

VI\_ATTR\_MEM\_SIZE: N/A Oh to FFFFFFFh for 32-bit applications

Oh to FFFFFFFFFFFFFFF for 64-bit applications

VI\_ATTR\_MEM\_SIZE\_32: Oh to FFFFFFFh

VI\_ATTR\_MEM\_SIZE\_64: Oh to FFFFFFFFFFFFFFF

VI\_ATTR\_MEM\_SIZE, VI\_ATTR\_MEM\_SIZE\_32, and VI\_ATTR\_MEM\_SIZE\_64 specify the size of memory requested by the device in VXIbus address space. If the value of VI\_ATTR\_MEM\_SPACE is VI\_A16\_SPACE, the value of this attribute is meaningless for the given VXI device.

#### **Related Topics**

INSTR Resource

VI\_ATTR\_MEM\_BASE/VI\_ATTR\_MEM\_BASE\_32/VI\_ATTR\_MEM\_BASE VI\_ATTR\_MEM\_SPACE

## VI\_ATTR\_MEM\_SPACE

## **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	VI_A16_SPACE (1) VI_A24_SPACE (2) VI_A32_SPACE	VI_A16_SPACE
		(3)	

VI\_ATTR\_MEM\_SPACE specifies the VXIbus address space used by the device. The three types are A16, A24, or A32 memory address space.

A VXI device with memory in A24 or A32 space also has registers accessible in the configuration section of A16 space. A VME device with memory in multiple address spaces requires one VISA resource for each address space used.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_MEM\_BASE/VI\_ATTR\_MEM\_BASE\_32/VI\_ATTR\_MEM\_BASE VI\_ATTR\_MEM\_SIZE/VI\_ATTR\_MEM\_SIZE\_32/VI\_ATTR\_MEM\_SIZE\_6/

# VI\_ATTR\_MODEL\_CODE

#### **Resource Classes**

GPIB-VXI INSTR, PXI INSTR, USB INSTR, USB RAW, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0h to FFFFh	N/A

VI\_ATTR\_MODEL\_CODE specifies the model code for the device.

For VXI resources, this refers to the VXI Model Code.

For PXI INSTR resources, if the subsystem PCI Vendor ID is nonzero, this refers to the subsystem Device ID. Otherwise, this refers to the Device ID.

For USB resources, this refers to the Product ID (PID).

#### **Related Topics**

INSTR Resource VI\_ATTR\_MANF\_ID VI\_ATTR\_MODEL\_NAME

## VI\_ATTR\_MODEL\_NAME

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, PXI INSTR, PXI BACKPLANE, USB INSTR, USB RAW, VXI INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

This string attribute is the model name of the device.

Note The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can be different between VISA implementations and/or revisions.

#### **Related Topics**

INSTR Resource VI\_ATTR\_MANF\_NAME VI\_ATTR\_MODEL\_CODE

## VI\_ATTR\_OPER\_NAME

### **Resource Classes**

VI\_EVENT\_IO\_COMPLETION, VI\_EVENT\_EXCEPTION

Access Privilege	Data Type	Range	Default
Read Only	ViString	N/A	N/A

VI\_ATTR\_OPER\_NAME contains the name of the operation generating this event.

#### **Related Topics**

VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_STATUS VI\_EVENT\_EXCEPTION VI\_EVENT\_IO\_COMPLETION

# VI\_ATTR\_PXI\_ACTUAL\_LWIDTH

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	-1, 1, 2, 4, 8, 16	N/A

VI\_ATTR\_PXI\_ACTUAL\_LWIDTH specifies the PCI Express link width negotiated between the PCI Express host controller and the device. A value of –1 indicates that the device is not a PXI/PCI Express device.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_BUS\_NUM

**Resource Classes** 

**PXI INSTR** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0 to 255	N/A

VI\_ATTR\_PXI\_BUS\_NUM specifies the PCI bus number of this device. **Related Topics** 

INSTR Resource

## VI\_ATTR\_PXI\_CHASSIS

**Resource Classes** 

PXI INSTR, PXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	-1, 0 to 255	N/A

VI\_ATTR\_PXI\_CHASSIS specifies the PXI chassis number of this device. A value of -1 means the chassis number is unknown.

### **Related Topics**

**INSTR Resource** 

## VI\_ATTR\_PXI\_DEST\_TRIG\_BUS

### **Resource Classes**

PXI BACKPLANE
Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	Single-Segment Chassis (8 Slots or Less): N/A	-1
		Multisegment Chassis (More than 8 Slots): 1number of chassis segments <u>*</u>	

VI\_ATTR\_PXI\_DEST\_TRIG\_BUS specifies the segment to use to qualify trigDest in viMapTrigger.

M **Note** Some PXI chassis, typically those with more than 8 slots, have multiple trigger buses (also called segments). viMapTrigger is used on the PXI BACKPLANE resource to map a trigger between two trigger buses. One trigger bus, specified by VI ATTR PXI SRC TRIG BUS, is the source or "writer" for this trigger line. The other trigger bus, specified by VI ATTR PXI DEST TRIG BUS, is a "reader." You can have multiple readers, but only one writer for a given trigger line. For example, if you want to have triggers mapped from trigger bus 1 to trigger bus 2 and then from trigger bus 2 to trigger bus 3, observe that in this case trigger bus 1 is the writer for this line, writing to both trigger bus 2 and trigger bus 3. Therefore, you should perform your viMapTrigger from 1 to 2 and from 1 to 3—mapping from 1 to 2 and then 2 to 3 would not be allowed because it would require 2 also to be a writer (as well as 1). Note also that mapping from one line in the source trigger bus to a different line in the destination trigger bus (trigSrc != trigDest) is dependent on hardware capabilities and a specific software implementation, and may not be supported.

Code to map trigger 5 from trigger segment 1 to trigger segment 2 of an 18-slot chassis would look like the following, where backplaneSession is a session to a PXI BACKPLANE resource:

viSetAttribute(backplaneSession, VI\_ATTR\_PXI\_SRC\_TRIG\_BUS, 1); viSetAttribute(backplaneSession, VI\_ATTR\_PXI\_DEST\_TRIG\_BUS, 2);

viMapTrigger(backplaneSession, VI\_TRIG\_TTL5, VI\_TRIG\_TTL5, VI\_NULL);

\*You can determine the number of segments from MAX (in the trigger reservation panel), from the chassis documentation, and by looking at the dividing lines on the physical front panel of the chassis itself.

#### **Related Topics**

BACKPLANE Resource VI\_ATTR\_PXI\_SRC\_TRIG\_BUS VI\_ATTR\_PXI\_TRIG\_BUS viMapTrigger

# VI\_ATTR\_PXI\_DEV\_NUM

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0 to 31	N/A

This is the PXI device number.

#### **Related Topics**

INSTR Resource

# VI\_ATTR\_PXI\_DSTAR\_BUS

## **Resource Classes**

Access Privilege	Data Type	Range		Default
Read Only Global	ViInt16		N/A	N/A

VI\_ATTR\_PXI\_DSTAR\_BUS specifies the differential star bus number of this device. A value of -1 means the chassis is unidentified or does not have a timing slot.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_DSTAR\_SET

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	-1, 0 to 16	N/A

VI\_ATTR\_PXI\_DSTAR\_SET specifies the set of PXI\_DSTAR lines connected to this device. A value of -1 means the chassis is unidentified or does not have a timing slot.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_FUNC\_NUM

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0 to 7	0

This is the PCI function number of the PXI/PCI resource. For most devices, the function number is 0, but a multifunction device may have a function number up to 7. The meaning of a function number other than 0 is device specific.

#### **Related Topics**

INSTR Resource

# VI\_ATTR\_PXI\_IS\_EXPRESS

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only	ViBoolean	VI_TRUE, VI_FALSE	N/A
Global			

VI\_ATTR\_PXI\_IS\_EXPRESS specifies whether the device is PXI/PCI or PXI/PCI Express.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_MAX\_LWIDTH

## **Resource Classes**

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	-1, 1, 2, 4, 8, 16	N/A

VI\_ATTR\_PXI\_MAX\_LWIDTH specifies the maximum PCI Express link width of the device. A value of -1 indicates that the device is not a PXI/PCI Express device.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_PXI\_MEM\_BASE\_BAR0/ VI\_ATTR\_PXI\_MEM\_BASE\_BAR1/ VI\_ATTR\_PXI\_MEM\_BASE\_BAR2/ VI\_ATTR\_PXI\_MEM\_BASE\_BAR3/ VI\_ATTR\_PXI\_MEM\_BASE\_BAR4/ VI\_ATTR\_PXI\_MEM\_BASE\_BAR5

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only	ViUInt32	0 to FFFFFFFh	N/A
Global			

PXI memory base address assigned to the specified BAR. If the value of the corresponding VI\_ATTR\_PXI\_MEM\_TYPE\_BAR*x* is VI\_PXI\_ADDR\_NONE, the value of this attribute is meaningless for the given PXI device.

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_MEM\_SIZE\_BAR0/1/2/3/4/5 VI\_ATTR\_PXI\_MEM\_TYPE\_BAR0/1/2/3/4/5 VI\_ATTR\_PXI\_MEM\_SIZE\_BAR0/ VI\_ATTR\_PXI\_MEM\_SIZE\_BAR1/ VI\_ATTR\_PXI\_MEM\_SIZE\_BAR2/ VI\_ATTR\_PXI\_MEM\_SIZE\_BAR3/ VI\_ATTR\_PXI\_MEM\_SIZE\_BAR4/ VI\_ATTR\_PXI\_MEM\_SIZE\_BAR5

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only	ViUInt32	0 to FFFFFFFh	N/A
Global			

Memory size used by the device in the specified BAR. If the value of the corresponding VI\_ATTR\_PXI\_MEM\_TYPE\_BAR*x* is VI\_PXI\_ADDR\_NONE, the value of this attribute is meaningless for the given PXI device.

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_MEM\_BASE\_BAR0/1/2/3/4/5 VI\_ATTR\_PXI\_MEM\_TYPE\_BAR0/1/2/3/4/5

# VI\_ATTR\_PXI\_MEM\_TYPE\_BAR0/ VI\_ATTR\_PXI\_MEM\_TYPE\_BAR1/ VI\_ATTR\_PXI\_MEM\_TYPE\_BAR2/ VI\_ATTR\_PXI\_MEM\_TYPE\_BAR3/ VI\_ATTR\_PXI\_MEM\_TYPE\_BAR4/ VI\_ATTR\_PXI\_MEM\_TYPE\_BAR5

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only	ViUInt16	VI_PXI_ADDR_NONE(0) VI_PXI_ADDR_MEM(1)	N/A
Global		VI_PXI_ADDR_IO(2)	

Memory type used by the device in the specified BAR (if applicable).

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_MEM\_BASE\_BAR0/1/2/3/4/5 VI\_ATTR\_PXI\_MEM\_SIZE\_BAR0/1/2/3/4/5

# VI\_ATTR\_PXI\_RECV\_INTR\_DATA

## **Resource Classes**

VI\_EVENT\_PXI\_INTR

Access Privilege	Data Type	Range	Default
Read Only	ViUInt32	N/A	N/A

VI\_ATTR\_PXI\_RECV\_INTR\_DATA shows the first PXI/PCI register that was read in the successful interrupt detection sequence.

#### **Related Topics**

VI\_ATTR\_PXI\_RECV\_INTR\_SEQ VI\_EVENT\_PXI\_INTR

# VI\_ATTR\_PXI\_RECV\_INTR\_SEQ

## **Resource Classes**

VI\_EVENT\_PXI\_INTR

Access Privilege	Data Type	Range		Default
Read Only	ViInt16		N/A	N/A

VI\_ATTR\_PXI\_RECV\_INTR\_SEQ shows the index of the interrupt sequence that detected the interrupt condition.

#### **Related Topics**

VI\_ATTR\_PXI\_RECV\_INTR\_DATA VI\_EVENT\_PXI\_INTR
# VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	VI_PXI_LBUS_UNKNOWN (-1); VI_PXI_LBUS_NONE (0); Normal slots (1 to 18); VI_PXI_LBUS_STAR_TRIG_BUS_0 (1000) to VI_PXI_LBUS_STAR_TRIG_BUS_9 (1009); VI_PXI_STAR_TRIG_CONTROLLER (1413); VI_PXI_LBUS_SCXI (2000)	N/A
		( )	

VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT specifies the slot number or special feature connected to the local bus left lines of this device.

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT VI\_ATTR\_SLOT

# VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	VI_PXI_LBUS_UNKNOWN (-1); VI_PXI_LBUS_NONE (0); Normal slots (1 to 18); VI_PXI_LBUS_STAR_TRIG_BUS_0 (1000) to VI_PXI_LBUS_STAR_TRIG_BUS_9 (1009); VI_PXI_STAR_TRIG_CONTROLLER (1413); VI_PXI_LBUS_SCXI (2000)	N/A
		( )	

VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT specifies the slot number or special feature connected to the local bus right lines of this device.

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT VI\_ATTR\_SLOT

# VI\_ATTR\_PXI\_SLOT\_LWIDTH

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	-1, 1, 4, 8	N/A

VI\_ATTR\_PXI\_SLOT\_LWIDTH specifies the PCI Express link width of the PXI Express peripheral slot in which the device resides. A value of -1 indicates that the device is not a PXI Express device.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_SLOTPATH

**Resource Classes** 

**PXI INSTR** 

Access Privilege	Data Type	Range		Default
Read Only Global	ViString		N/A	N/A

VI\_ATTR\_PXI\_SLOTPATH specifies the slot path of this device.

The purpose of a PXI slot path is to describe the PCI bus hierarchy in a manner independent of the PCI bus number. PXI slot paths are a sequence of values representing the PCI device number and function number of a PCI module and each parent PCI bridge that routes the module to the host PCI bridge (bus 0). Each value is represented as "dev[.func]", where the function number is listed only if it is non-zero. When a PXI slot path includes multiple values, the values are commaseparated.

The string format of the attribute value looks like this:

device1[.function1][,device2[.function2]][,...]

An example string is "5.1,12,8". In this case, there is a PCI-to-PCI bridge on device 8 on the root bus. On its secondary bus, there is another PCIto-PCI bridge on device 12. On its secondary bus, there is an instrument on device 5, function 1. The example string value describes this instrument's slot path.

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_PXI\_SRC\_TRIG\_BUS

#### **Resource Classes**

PXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	Single-Segment Chassis (8 Slots or Less): N/A	-1
		Multisegment Chassis (More than 8 Slots): 1number of chassis segments <u>*</u>	

VI\_ATTR\_PXI\_SRC\_TRIG\_BUS specifies the segment to use to qualify trigSrc in viMapTrigger.

M **Note** Some PXI chassis, typically those with more than 8 slots, have multiple trigger buses (also called segments). viMapTrigger is used on the PXI BACKPLANE resource to map a trigger between two trigger buses. One trigger bus, specified by VI ATTR PXI SRC TRIG BUS, is the source or "writer" for this trigger line. The other trigger bus, specified by VI ATTR PXI DEST TRIG BUS, is a "reader." You can have multiple readers, but only one writer for a given trigger line. For example, if you want to have triggers mapped from trigger bus 1 to trigger bus 2 and then from trigger bus 2 to trigger bus 3, observe that in this case trigger bus 1 is the writer for this line, writing to both trigger bus 2 and trigger bus 3. Therefore, you should perform your viMapTrigger from 1 to 2 and from 1 to 3—mapping from 1 to 2 and then 2 to 3 would not be allowed because it would require 2 also to be a writer (as well as 1). Note also that mapping from one line in the source trigger bus to a different line in the destination trigger bus (trigSrc != trigDest) is dependent on hardware capabilities and a specific software implementation, and may not be supported.

Code to map trigger 5 from trigger segment 1 to trigger segment 2 of an 18-slot chassis would look like the following, where backplaneSession is a session to a PXI BACKPLANE resource:

viSetAttribute(backplaneSession, VI\_ATTR\_PXI\_SRC\_TRIG\_BUS, 1); viSetAttribute(backplaneSession, VI\_ATTR\_PXI\_DEST\_TRIG\_BUS, 2);

viMapTrigger(backplaneSession, VI\_TRIG\_TTL5, VI\_TRIG\_TTL5, VI\_NULL);

\*You can determine the number of segments from MAX (in the trigger reservation panel), from the chassis documentation, and by looking at the dividing lines on the physical front panel of the chassis itself.

### **Related Topics**

BACKPLANE Resource VI\_ATTR\_PXI\_DEST\_TRIG\_BUS VI\_ATTR\_PXI\_TRIG\_BUS viMapTrigger

# VI\_ATTR\_PXI\_STAR\_TRIG\_BUS

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range		Default
Read Only Global	ViInt16		N/A	N/A

VI\_ATTR\_PXI\_STAR\_TRIG\_BUS specifies the star trigger bus number of this device.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_PXI\_STAR\_TRIG\_LINE

VI\_ATTR\_PXI\_TRIG\_BUS

# VI\_ATTR\_PXI\_STAR\_TRIG\_LINE

### **Resource Classes**

**PXI INSTR** 

Access Privilege	Data Type	Range		Default
Read Only Global	ViInt16		N/A	N/A

VI\_ATTR\_PXI\_STAR\_TRIG\_LINE specifies the PXI\_STAR line connected to this device.

#### **Related Topics**

**INSTR Resource** 

VI\_ATTR\_PXI\_STAR\_TRIG\_BUS

VI\_ATTR\_PXI\_TRIG\_BUS

# VI\_ATTR\_PXI\_TRIG\_BUS

### **Resource Classes**

PXI INSTR, PXI BACKPLANE

Access Privilege	Data Type	Range	Default
INSTR: Read Only Global	ViInt16	N/A	N/A
BACKPLANE: Read/Write Local			

VI\_ATTR\_PXI\_TRIG\_BUS specifies the trigger bus number of this device.

#### **Related Topics**

INSTR Resource VI\_ATTR\_PXI\_STAR\_TRIG\_BUS VI\_ATTR\_PXI\_STAR\_TRIG\_LINE

# VI\_ATTR\_RD\_BUF\_OPER\_MODE

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_FLUSH_ON_ACCESS (1) VI_FLUSH_DISABLE (3)	VI_FLUSH_DISABLE

VI\_ATTR\_RD\_BUF\_OPER\_MODE specifies the operational mode of the formatted I/O read buffer. When the operational mode is set to VI\_FLUSH\_DISABLE (default), the buffer is flushed only on explicit calls to viFlush(). If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the read buffer is flushed every time a viScanf() (or related) operation completes.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_WR\_BUF\_OPER\_MODE viFlush viScanf

# VI\_ATTR\_RD\_BUF\_SIZE

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read Only Local	ViUInt32	N/A	N/A

This is the current size of the formatted I/O input buffer for this session. The user can modify this value by calling viSetBuf().

#### **Related Topics**

VI\_ATTR\_RD\_BUF\_OPER\_MODE VI\_ATTR\_WR\_BUF\_SIZE viSetBuf

# VI\_ATTR\_RECV\_INTR\_LEVEL

## **Resource Classes**

VI\_EVENT\_VXI\_VME\_INTR

Access Privilege	Data Type	Range	Default
Read Only	ViInt16	1 to 7;VI_UNKNOWN_LEVEL	N/A
		(-1)	

VI\_ATTR\_RECV\_INTR\_LEVEL is the VXI interrupt level on which the interrupt was received.

#### **Related Topics**

INSTR Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_INTR\_STATUS\_ID VI\_EVENT\_VXI\_VME\_INTR

# VI\_ATTR\_RECV\_TRIG\_ID

**Resource Classes** 

VI\_EVENT\_TRIG

ant i
J/A
VI\_ATTR\_RECV\_TRIG\_ID identifies the triggering mechanism on which the specified trigger event was received.

#### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource SERVANT Resource VI\_EVENT\_TRIG

# VI\_ATTR\_RET\_COUNT/VI\_ATTR\_RET\_COUNT\_32

**Resource Classes** 

VI\_EVENT\_IO\_COMPLETION

Access Privilege Data Type

Range

Read VI\_ATTR\_RET\_COUNT: Only ViUInt32 for 32-bit applications

ViUInt64 for 64-bit applications

applications 0h to

**VI ATTR RET COUNT:** 

Oh to FFFFFFFh for 32-bit

FFFFFFFFFFFFFFF for 64-bit applications

VI\_ATTR\_RET\_COUNT\_32: ViUInt32 VI\_ATTR

VI\_ATTR\_RET\_COUNT\_32: Oh to FFFFFFFh

VI\_ATTR\_RET\_COUNT\_64:

ViUInt64

VI\_ATTR\_RET\_COUNT\_64: Oh to FFFFFFFFFFFFFFFF

De

Ν

VI\_ATTR\_RET\_COUNT, VI\_ATTR\_RET\_COUNT\_32, and VI\_ATTR\_RET\_COUNT\_64 contain the actual number of elements that were asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_32 is always a 32-bit value.

VI\_ATTR\_RET\_COUNT\_64 is always a 64-bit value. VI\_ATTR\_RET\_COUNT\_64 is not supported with 32-bit applications.

VI\_ATTR\_RET\_COUNT is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications.

#### **Related Topics**

VI\_ATTR\_BUFFER VI\_ATTR\_JOB\_ID VI\_ATTR\_STATUS VI\_EVENT\_IO\_COMPLETION

# VI\_ATTR\_RM\_SESSION

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read Only Local	ViSession	N/A	N/A

VI\_ATTR\_RM\_SESSION specifies the session of the Resource Manager that was used to open this session.

#### **Related Topics**

VISA Resource Template

# VI\_ATTR\_RSRC\_CLASS

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

VI\_ATTR\_RSRC\_CLASS specifies the resource class (for example, "INSTR") as defined by the canonical resource name.

#### **Related Topics**

VI\_ATTR\_RSRC\_NAME VISA Resource Template

# VI\_ATTR\_RSRC\_IMPL\_VERSION

# **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

Access Privilege	Data Type	Range	Default
Read Only	ViVersion	0h to FFFFFFFh	N/A
Global			

VI\_ATTR\_RSRC\_IMPL\_VERSION is the resource version that uniquely identifies each of the different revisions or implementations of a resource. This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

#### **Related Topics**

VI\_ATTR\_RSRC\_SPEC\_VERSION VISA Resource Template

# VI\_ATTR\_RSRC\_LOCK\_STATE

### **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read	ViAccessMode	VI_NO_LOCK (0)	VI_NO_LOCK
Only		VI_EXCLUSIVE_LOCK	
Global		(1)	
		VI_SHARED_LOCK (2)	

VI\_ATTR\_RSRC\_LOCK\_STATE indicates the current locking state of the resource. The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

#### **Related Topics**

VISA Resource Template

# VI\_ATTR\_RSRC\_MANF\_ID

# **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0h to 3FFFh	N/A

VI\_ATTR\_RSRC\_MANF\_ID is a value that corresponds to the VXI manufacturer ID of the vendor that implemented the VISA library. This attribute is not related to the device manufacturer attributes.

### **Related Topics**

VI\_ATTR\_RSRC\_MANF\_NAME VISA Resource Template

# VI\_ATTR\_RSRC\_MANF\_NAME

## **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

VI\_ATTR\_RSRC\_MANF\_NAME is a string that corresponds to the manufacturer name of the vendor that implemented the VISA library. This attribute is not related to the device manufacturer attributes.



**Note** The value of this attribute is for display purposes only and not for programmatic decisions, as the value can differ between VISA implementations and/or revisions.

#### **Related Topics**

VI\_ATTR\_RSRC\_MANF\_ID VISA Resource Template

# VI\_ATTR\_RSRC\_NAME

## **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read Only Global	ViRsrc	N/A	N/A

VI\_ATTR\_RSRC\_NAME is the unique identifier for a resource. Refer to <u>VISA Resource Syntax and Examples</u> for the syntax of resource strings and examples.

#### **Related Topics**

viFindRsrc viOpen viParseRsrc VISA Resource Template

# VI\_ATTR\_RSRC\_SPEC\_VERSION

## **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

Access Privilege	Data Type	Range	Default
Read Only Global	ViVersion	0h to FFFFFFFh	00300000h

VI\_ATTR\_RSRC\_SPEC\_VERSION is the resource version that uniquely identifies the version of the VISA specification to which the implementation is compliant. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version. The current VISA specification defines the value to be 00300000h.

#### **Related Topics**

VI\_ATTR\_RSRC\_IMPL\_VERSION VISA Resource Template

# VI\_ATTR\_SEND\_END\_EN

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, USB INSTR, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_TRUE

VI\_ATTR\_SEND\_END\_EN specifies whether to assert END during the transfer of the last byte of the buffer.

VI\_ATTR\_SEND\_END\_EN is relevant only in viWrite and related operations.

On Serial INSTR sessions, if this attribute is set to VI\_FALSE, the write will transmit the exact contents of the user buffer, without modifying it and without appending anything to the data being written. If this attribute is set to VI\_TRUE, VISA will perform the behavior described in VI\_ATTR\_ASRL\_END\_OUT.

On GPIB, VXI, GPIB-VXI, TCP/IP INSTR, and USB INSTR sessions, if this attribute is set to VI\_TRUE, VISA will include the 488.2 defined "end of message" terminator.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_ASRL\_END\_OUT viWrite

# VI\_ATTR\_SIGP\_STATUS\_ID

### **Resource Classes**

VI\_EVENT\_VXI\_SIGP

Access Privilege	Data Type	Range	Default
Read Only	ViUInt16	0h to FFFFh	N/A

VI\_ATTR\_SIGP\_STATUS\_ID is the 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.

#### **Related Topics**

**INSTR Resource** 

VI\_EVENT\_VXI\_SIGP

# VI\_ATTR\_SLOT

### **Resource Classes**

GPIB-VXI INSTR, PXI INSTR, VXI INSTR

Access Privilege	Data Type	Range	Default
Read	ViInt16	VXI	N/A
Only Global		0 to 12 VI_UNKNOWN_SLOT (-1) <b>PXI</b>	
		1 to 18 VI_UNKNOWN_SLOT (-1)	
VI\_ATTR\_SLOT specifies the physical slot location of the device. If the slot number is not known, VI\_UNKNOWN\_SLOT is returned.

### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_SRC\_ACCESS\_PRIV

#### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_DATA_PRIV (0)	VI_DATA_PRIV
		VI_DATA_NPRIV	
		(1)	
		VI_PROG_PRIV	
		(2)	
		VI_PROG_NPRIV	
		(3)	
		VI_BLCK_PRIV	
		(4)	
		VI_BLCK_NPRIV	
		(5)	
		VI_D64_PRIV (6)	
		VI_D64_NPRIV	
		(7)	

VI\_ATTR\_SRC\_ACCESS\_PRIV specifies the address modifier to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.



**Note** The values VI\_D64\_PRIV (6) and VI\_D64\_NPRIV (7) apply to only the block move operations. If you set this attribute to one of these values and then call one of the viIn*XX*() operations, the operation returns VI\_ERROR\_INV\_SETUP.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_ACCESS\_PRIV VI\_ATTR\_SRC\_BYTE\_ORDER VI\_ATTR\_SRC\_INCREMENT VI\_ATTR\_WIN\_ACCESS\_PRIV

# VI\_ATTR\_SRC\_BYTE\_ORDER

#### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range		C	Default	
Read/Write Local	ViUInt16	VI_BIG_ VI_LITT	_ENDIAN ( `LE_ENDIA (1)	(0) V AN	/I_BIG_	ENDIAN

VI\_ATTR\_SRC\_BYTE\_ORDER specifies the byte order to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_BYTE\_ORDER VI\_ATTR\_SRC\_ACCESS\_PRIV VI\_ATTR\_SRC\_INCREMENT VI\_ATTR\_WIN\_BYTE\_ORDER

## VI\_ATTR\_SRC\_INCREMENT

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt32	0 to 1	1

VI\_ATTR\_SRC\_INCREMENT is used in the viMoveInXX() operations to specify by how many elements the source offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveInXX() operations move from consecutive elements. If this attribute is set to 0, the viMoveInXX() operations will always read from the same element, essentially treating the source as a FIFO register.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_SRC\_ACCESS\_PRIV VI\_ATTR\_SRC\_BYTE\_ORDER

## VI\_ATTR\_STATUS

### **Resource Classes**

VI\_EVENT\_EXCEPTION, VI\_EVENT\_IO\_COMPLETION, VI\_EVENT\_USB\_INSTR

Access Privilege	Data Type	Range	Default
Read Only	ViStatus	N/A	N/A

VI\_ATTR\_STATUS contains the return code of the operation generating this event.

### **Related Topics**

VI\_ATTR\_BUFFER VI\_ATTR\_JOB\_ID VI\_ATTR\_MODEL\_NAME VI\_ATTR\_OPER\_NAME VI\_ATTR\_RET\_COUNT/VI\_ATTR\_RET\_COUNT\_32/VI\_ATTR\_RET\_COI VI\_EVENT\_EXCEPTION VI\_EVENT\_IO\_COMPLETION

VI\_EVENT\_USB\_INTR

## VI\_ATTR\_SUPPRESS\_END\_EN

### **Resource Classes**

Serial INSTR, TCPIP SOCKET, USB RAW, VXI INSTR

\_

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

VI\_ATTR\_SUPPRESS\_END\_EN is relevant only in viRead and related operations.

For all session types on which this attribute is supported, if this attribute is set to VI\_TRUE, read will not terminate due to an END condition. However, a read may still terminate successfully if VI\_ATTR\_TERMCHAR\_EN is set to VI\_TRUE. Otherwise, read will not

terminate until all requested data is received (or an error occurs).

On Serial INSTR sessions, if this attribute is set to VI\_FALSE, VISA will perform the behavior described in VI\_ATTR\_ASRL\_END\_IN.

On USB RAW sessions, if this attribute is set to VI\_FALSE, VISA will perform the behavior described in VI\_ATTR\_USB\_END\_IN.

On TCP/IP SOCKET sessions, if this attribute is set to VI\_FALSE, if NI-VISA reads some data and then detects a pause in the arrival of data packets, it will terminate the read operation. On TCP/IP SOCKET sessions, this attribute defaults to VI\_TRUE in NI-VISA.

On VXI INSTR sessions, if this attribute is set to VI\_FALSE, the END bit terminates read operations.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_END\_IN VI\_ATTR\_USB\_END\_IN viRead

# VI\_ATTR\_TCPIP\_ADDR

## **Resource Classes**

TCPIP INSTR, TCPIP SOCKET

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

This is the TCPIP address of the device to which the session is connected. This string is formatted in dot notation.

#### **Related Topics**

INSTR Resource SOCKET Resource VI\_ATTR\_TCPIP\_HOSTNAME

# VI\_ATTR\_TCPIP\_DEVICE\_NAME

### **Resource Classes**

TCPIP INSTR

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

This specifies the LAN device name used by the VXI-11 or LXI protocol during connection.

### **Related Topics**

INSTR Resource

# VI\_ATTR\_TCPIP\_HOSTNAME

## **Resource Classes**

TCPIP INSTR, TCPIP SOCKET

Access Privilege	Data Type	Range	Default
Read Only Global	ViString	N/A	N/A

This specifies the host name of the device. If no host name is available, this attribute returns an empty string.

#### **Related Topics**

INSTR Resource SOCKET Resource VI\_ATTR\_TCPIP\_ADDR

# VI\_ATTR\_TCPIP\_KEEPALIVE

**Resource Classes** 

TCPIP SOCKET

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE(1) VI_FALSE(0)	VI_FALSE

Setting this attribute to TRUE requests that a TCP/IP provider enable the use of keep-alive packets on TCP connections. After the system detects that a connection was dropped, VISA returns a lost connection error code on subsequent I/O calls on the session. The time required for the system to detect that the connection was dropped is dependent on the system and is not settable.

#### **Related Topics**

SOCKET Resource VI\_ATTR\_TCPIP\_NODELAY

# VI\_ATTR\_TCPIP\_NODELAY

**Resource Classes** 

TCPIP SOCKET

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE(1) VI_FALSE(0)	VI_TRUE

The Nagle algorithm is disabled when this attribute is enabled (and vice versa). The Nagle algorithm improves network performance by buffering "send" data until a full-size packet can be sent. This attribute is enabled by default in VISA to verify that synchronous writes get flushed immediately.

#### **Related Topics**

SOCKET Resource VI\_ATTR\_TCPIP\_KEEPALIVE

# VI\_ATTR\_TCPIP\_PORT

**Resource Classes** 

TCPIP SOCKET

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	0 to FFFFh	N/A

This specifies the port number for a given TCPIP address. For a TCPIP SOCKET Resource, this is a required part of the address string.

### **Related Topics**

SOCKET Resource

## VI\_ATTR\_TERMCHAR

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt8	0 to FFh	0Ah (linefeed)
VI\_ATTR\_TERMCHAR is the termination character. When the termination character is read and VI\_ATTR\_TERMCHAR\_EN is enabled during a read operation, the read operation terminates.

For a Serial INSTR session, VI\_ATTR\_TERMCHAR is Read/Write when the corresponding session is not enabled to receive VI\_EVENT\_ASRL\_TERMCHAR events. When the session is enabled to receive VI\_EVENT\_ASRL\_TERMCHAR events, the attribute VI\_ATTR\_TERMCHAR is Read Only. For all other session types, the attribute VI\_ATTR\_TERMCHAR is always Read/Write.

#### **Related Topics**

**INSTR Resource** 

**INTFC Resource** 

SERVANT Resource

SOCKET Resource

VI\_ATTR\_TERMCHAR\_EN

## VI\_ATTR\_TERMCHAR\_EN

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

\_

Access Privilege	Data Type	Range	Default
Read/Write Local	ViBoolean	VI_TRUE (1) VI_FALSE (0)	VI_FALSE

VI\_ATTR\_TERMCHAR\_EN is a flag that determines whether the read operation should terminate when a termination character is received. This attribute is valid for both raw I/O (viRead) and formatted I/O (viScanf).

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_TERMCHAR

# VI\_ATTR\_TMO\_VALUE

### **Resource Classes**

All I/O session types

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt32	VI_TMO_IMMEDIATE (0); 1 to FFFFFFFh; VI_TMO_INFINITE (FFFFFFh)	2000

VI\_ATTR\_TMO\_VALUE specifies the minimum timeout value to use (in milliseconds) when accessing the device associated with the given session. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

Notice that the actual timeout value used by the driver may be higher than the requested one. The actual timeout value is returned when this attribute is retrieved via viGetAttribute().

#### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource MEMACC Resource Resources SERVANT Resource SOCKET Resource

## VI\_ATTR\_TRIG\_ID

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI BACKPLANE, GPIB-VXI INSTR, PXI INSTR, PXI BACKPLANE, Serial INSTR, TCPIP INSTR, VXI BACKPLANE, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	GPIB, Serial, TCPIP: VI_TRIG_SW (- 1)	VI_TRIG_SW
		VXI, GPIB-VXI: VI_TRIG_SW (- 1); VI_TRIG_TTL0 (0) to VI_TRIG_TTL7 (7); VI_TRIG_ECL0 (8) to VI_TRIG_ECL1 (9)	VI_TRIG_SW
		PXI INSTR, PXI BACKPLANE: VI_TRIG_SW (-1) VI_TRIG_TTLO (0) to VI_TRIG_TTL7 (7)	VI_TRIG_SW

VI\_ATTR\_TRIG\_ID is the identifier for the current triggering mechanism.

VI\_ATTR\_TRIG\_ID is Read/Write when the corresponding session is not enabled to receive trigger events. When the session is enabled to receive trigger events, the attribute VI\_ATTR\_TRIG\_ID is Read Only.

#### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource SERVANT Resource VI\_ATTR\_RECV\_TRIG\_ID viAssertTrigger

# VI\_ATTR\_USB\_ALT\_SETTING

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Global	ViInt16	0 to FFh	0

VI\_ATTR\_USB\_ALT\_SETTING specifies the USB alternate setting used by this USB interface.

VI\_ATTR\_USB\_ALT\_SETTING is Read/Write when the corresponding session is not enabled to receive USB interrupt events. If the session is enabled to receive USB interrupt events or if there are any other sessions to this resource, the attribute VI\_ATTR\_USB\_ALT\_SETTING is Read Only.

#### **Related Topics**

RAW Resource

# VI\_ATTR\_USB\_BULK\_IN\_PIPE

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	-1, 81h to 8Fh	N/A

VI\_ATTR\_USB\_BULK\_IN\_PIPE specifies the endpoint address of the USB bulk-in pipe used by the given session. An initial value of -1 signifies that this resource does not have any bulk-in pipes. This endpoint is used in viRead and related operations.

#### **Related Topics**

RAW Resource

VI\_ATTR\_USB\_BULK\_OUT\_PIPE

VI\_ATTR\_USB\_CTRL\_PIPE

VI\_ATTR\_USB\_INTR\_IN\_PIPE

VI\_ATTR\_USB\_NUM\_PIPES

# VI\_ATTR\_USB\_BULK\_IN\_STATUS

### **Resource Classes**

Read/Write ViInt16 VI_USB_PIPE_STATE_UNKNOWN	
Local (-1) VI_USB_PIPE_READY (0)	J/A
VI_USB_PIPE_STALLED (1)	

VI\_ATTR\_USB\_BULK\_IN\_STATUS specifies whether the USB bulk-in pipe used by the given session is stalled or ready. This attribute can be set to only VI\_USB\_PIPE\_READY.

#### **Related Topics**

RAW Resource

# VI\_ATTR\_USB\_BULK\_OUT\_PIPE

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	-1, 01h to 0Fh	N/A

VI\_ATTR\_USB\_BULK\_OUT\_PIPE specifies the endpoint address of the USB bulk-out or interrupt-out pipe used by the given session. An initial value of -1 signifies that this resource does not have any bulk-out or interrupt-out pipes. This endpoint is used in viWrite and related operations.

#### **Related Topics**

RAW Resource VI\_ATTR\_USB\_BULK\_IN\_PIPE VI\_ATTR\_USB\_CTRL\_PIPE VI\_ATTR\_USB\_INTR\_IN\_PIPE VI\_ATTR\_USB\_NUM\_PIPES

# VI\_ATTR\_USB\_BULK\_OUT\_STATUS

### **Resource Classes**

Read/Write ViInt16 VI_USB_PIPE_STATE_UNKNOWN	
Local (-1) VI_USB_PIPE_READY (0)	J/A
VI_USB_PIPE_STALLED (1)	

VI\_ATTR\_USB\_BULK\_OUT\_STATUS specifies whether the USB bulk-out or interrupt-out pipe used by the given session is stalled or ready. This attribute can be set to only VI\_USB\_PIPE\_READY.

#### **Related Topics**

**RAW Resource** 

# VI\_ATTR\_USB\_CLASS

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to FFh	N/A

 $\rm VI\_ATTR\_USB\_CLASS$  specifies the USB class used by this USB interface.

#### **Related Topics**

RAW Resource

# VI\_ATTR\_USB\_CTRL\_PIPE

### **Resource Classes**

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	00h to 0Fh	00h

VI\_ATTR\_USB\_CTRL\_PIPE specifies the endpoint address of the USB control pipe used by the given session. A value of 0 signifies that the default control pipe will be used. This endpoint is used in viUsbControlIn and viUsbControlOut operations. Nonzero values may not be supported on all platforms.

#### **Related Topics**

RAW Resource VI\_ATTR\_USB\_BULK\_IN\_PIPE VI\_ATTR\_USB\_BULK\_OUT\_PIPE VI\_ATTR\_USB\_INTR\_IN\_PIPE VI\_ATTR\_USB\_NUM\_PIPES viUsbControlIn viUsbControlOut

# VI\_ATTR\_USB\_END\_IN

**Resource Classes** 

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_USB_END_NONE (0) VI_USB_END_SHORT (4)	VI_USB_I
		VI_USB_END_SHORT_OR_COUNT (5)	

VI\_ATTR\_USB\_END\_IN indicates the method used to terminate read operations.

If it is set to VI\_USB\_END\_NONE, short packets are ignored for read operations, so reads will not terminate until all of the requested data is received (or an error occurs).

If it is set to VI\_USB\_END\_SHORT, the read operation will terminate on a short packet; use this if the device will terminate all read transfers with a short packet, including sending a zero (short) packet when the last data packet is full.

If it is set to VI\_USB\_END\_SHORT\_OR\_COUNT, the read operation will terminate on a short packet or when it receives the requested count of data bytes; use this if the device does not send zero packets.

#### **Related Topics**

RAW Resource

# VI\_ATTR\_USB\_INTFC\_NUM

### **Resource Classes**

USB INSTR, USB RAW

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to FEh	0
VI\_ATTR\_USB\_INTFC\_NUM specifies the USB interface number used by the given session.

### **Related Topics**

**INSTR Resource** 

RAW Resource

# VI\_ATTR\_USB\_INTR\_IN\_PIPE

### **Resource Classes**

USB RAW

Access Privilege	Data Type	Range	Default
Read/Write Local	ViInt16	-1, 81h to 8Fh	N/A

VI\_ATTR\_USB\_INTR\_IN\_PIPE specifies the endpoint address of the USB interrupt-in pipe used by the given session. An initial value of -1 signifies that this resource does not have any interrupt-in pipes. This endpoint is used in viEnableEvent for VI\_EVENT\_USB\_INTR.

### **Related Topics**

RAW Resource VI\_ATTR\_USB\_BULK\_IN\_PIPE VI\_ATTR\_USB\_BULK\_OUT\_PIPE VI\_ATTR\_USB\_CTRL\_PIPE VI\_ATTR\_USB\_NUM\_PIPES

# VI\_ATTR\_USB\_INTR\_IN\_STATUS

### **Resource Classes**

USB RAW

Read/Write ViInt16 VI_USB_PIPE_STATE_UNKNOWN N Local (-1) VI_USB_PIPE_READY (0)	ault
	/A
VI_USB_PIPE_STALLED (1)	

VI\_ATTR\_USB\_INTR\_IN\_STATUS specifies whether the USB interrupt-in pipe used by the given session is stalled or ready. This attribute can be set to only VI\_USB\_PIPE\_READY.

### **Related Topics**

RAW Resource

## VI\_ATTR\_USB\_MAX\_INTR\_SIZE

### **Resource Classes**

USB INSTR, USB RAW

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	0 to FFFFh	N/A

VI\_ATTR\_USB\_MAX\_INTR\_SIZE specifies the maximum size of data that will be stored by any given USB interrupt. If a USB interrupt contains more data than this size, the data in excess of this size will be lost.

VI\_ATTR\_USB\_MAX\_INTR\_SIZE is Read/Write when the corresponding session is not enabled to receive USB interrupt events. When the session is enabled to receive USB interrupt events, the attribute VI\_ATTR\_USB\_MAX\_INTR\_SIZE is Read Only.

#### **Related Topics**

**INSTR Resource** 

RAW Resource

VI\_EVENT\_USB\_INTR

# VI\_ATTR\_USB\_NUM\_INTFCS

### **Resource Classes**

USB RAW

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	1 to FFh	N/A

VI\_ATTR\_USB\_NUM\_INTFCS specifies the number of interfaces supported by this USB device.

### **Related Topics**

RAW Resource

# VI\_ATTR\_USB\_NUM\_PIPES

### **Resource Classes**

USB RAW

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to 30	N/A

VI\_ATTR\_USB\_NUM\_PIPES specifies the number of pipes supported by this USB interface. This does not include the default control pipe.

### **Related Topics**

RAW Resource

VI\_ATTR\_USB\_BULK\_IN\_PIPE

VI\_ATTR\_USB\_BULK\_OUT\_PIPE

VI\_ATTR\_USB\_INTR\_IN\_PIPE

## VI\_ATTR\_USB\_PROTOCOL

### **Resource Classes**

USB INSTR, USB RAW

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to FFh	N/A

VI\_ATTR\_USB\_PROTOCOL specifies the USB protocol used by this USB interface.

### **Related Topics**

**INSTR Resource** 

RAW Resource

## VI\_ATTR\_USB\_RECV\_INTR\_DATA

### **Resource Classes**

VI\_EVENT\_USB\_INTR

Access Privilege	Data Type	Range	Default
Read Only	ViAUInt8	N/A	N/A

VI\_ATTR\_USB\_RECV\_INTR\_DATA contains the actual received data from the USB Interrupt. The passed in data buffer *must* be of size at least equal to the value of VI\_ATTR\_USB\_RECV\_INTR\_SIZE.

#### **Related Topics**

VI\_ATTR\_USB\_RECV\_INTR\_SIZE VI\_EVENT\_USB\_INTR

## VI\_ATTR\_USB\_RECV\_INTR\_SIZE

### **Resource Classes**

VI\_EVENT\_USB\_INTR

Access Privilege	Data Type	Range	Default
Read Only	ViUInt16	N/A	N/A

VI\_ATTR\_USB\_RECV\_INTR\_SIZE contains the number of bytes of USB interrupt data that is stored.

### **Related Topics**

VI\_ATTR\_USB\_RECV\_INTR\_DATA VI\_EVENT\_USB\_INTR

# VI\_ATTR\_USB\_SERIAL\_NUM

### **Resource Classes**

USB INSTR, USB RAW

Access Privilege	Data Type	Range		Default
Read Only Global	ViString	I	N/A	N/A

VI\_ATTR\_USB\_SERIAL\_NUM specifies the USB serial number of this device.

### **Related Topics**

**INSTR Resource** 

RAW Resource

# VI\_ATTR\_USB\_SUBCLASS

**Resource Classes** 

USB RAW

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to FFh	N/A

VI\_ATTR\_USB\_SUBCLASS specifies the USB subclass used by this USB interface.

### **Related Topics**

RAW Resource

## VI\_ATTR\_USER\_DATA/VI\_ATTR\_USER\_DATA\_32

### **Resource Classes**

All I/O session types

Access **Data Type** Privilege

#### Range

Not specified

Read/Write VI\_ATTR\_USER\_DATA: VI\_ATTR\_USER\_DATA: Local ViAddr

> VI\_ATTR\_USER\_DATA\_32: VI\_ATTR\_USER\_DATA\_32: Oh to FFFFFFFh ViUInt32

> VI ATTR USER DATA 64: VI ATTR USER DATA 64: ViUInt64 Oh to FFFFFFFFFFFFF

D

VI\_ATTR\_USER\_DATA, VI\_ATTR\_USER\_DATA\_32, and VI\_ATTR\_USER\_DATA\_64 store data to be used privately by the application for a particular session. VISA does not use this data for any purpose. It is provided to the application for its own use.

VI\_ATTR\_USER\_DATA\_64 is not supported with 32-bit applications.

#### **Related Topics**

VISA Resource Template

# VI\_ATTR\_VXI\_DEV\_CLASS

### **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

Access Da Privilege Typ	ta Range pe	Default
Read Vil Only Global	JInt16 VI_VXI_CLAS VI_VXI_CLASS VI_VXI_CLASS VI_VXI_CLASS VI_VXI_CLASS	S_MEMORY(0) N/A S_EXTENDED(1) S_MESSAGE(2) S_REGISTER(3) SS_OTHER(4)
This attribute represents the VXI-defined device class to which the resource belongs, either message based (VI\_VXI\_CLASS\_MESSAGE), register based (VI\_VXI\_CLASS\_REGISTER), extended (VI\_VXI\_CLASS\_EXTENDED), or memory (VI\_VXI\_CLASS\_MEMORY). VME devices are usually either register based or belong to a miscellaneous class (VI\_VXI\_CLASS\_OTHER).

#### **Related Topics**

**INSTR Resource** 

# VI\_ATTR\_VXI\_LA

### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read Only Global	ViInt16	0 to 511	N/A

For an INSTR session, VI\_ATTR\_VXI\_LA specifies the logical address of the VXI or VME device used by the given session. For a MEMACC or SERVANT session, this attribute specifies the logical address of the local controller.

#### **Related Topics**

INSTR Resource MEMACC Resource SERVANT Resource

# VI\_ATTR\_VXI\_TRIG\_STATUS

### **Resource Classes**

GPIB-VXI BACKPLANE, VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt32	N/A	N/A

This attribute shows the current state of the VXI trigger lines. This is a bit vector with bits 0-9 corresponding to VI\_TRIG\_TTL0 through VI\_TRIG\_ECL1.

#### **Related Topics**

BACKPLANE Resource VI\_ATTR\_VXI\_TRIG\_SUPPORT VI\_ATTR\_VXI\_VME\_INTR\_STATUS VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE

# VI\_ATTR\_VXI\_TRIG\_SUPPORT

### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI BACKPLANE, VXI INSTR, VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt32	N/A	N/A

This attribute shows which VXI trigger lines this implementation supports. This is a bit vector with bits 0-9 corresponding to VI\_TRIG\_TTL0 through VI\_TRIG\_ECL1.

#### **Related Topics**

BACKPLANE Resource

**INSTR Resource** 

# VI\_ATTR\_VXI\_VME\_INTR\_STATUS

### **Resource Classes**

GPIB-VXI BACKPLANE, VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read Only Global	ViUInt16	N/A	N/A

This attribute shows the current state of the VXI/VME interrupt lines. This is a bit vector with bits 0-6 corresponding to interrupt lines 1-7.

### **Related Topics**

BACKPLANE Resource VI\_ATTR\_VXI\_TRIG\_STATUS VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE

# VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE

### **Resource Classes**

GPIB-VXI BACKPLANE, VXI BACKPLANE

Access Privilege	Data Type	Range	Default
Read	ViInt16	VI_STATE_ASSERTED(1)	N/A
Only		VI_STATE_DEASSERTED(0)	
Global		VI_STATE_UNKNOWN(-1)	

This attribute shows the current state of the VXI/VME SYSFAIL (SYStem FAILure) backplane line.

### **Related Topics**

BACKPLANE Resource VI\_ATTR\_VXI\_TRIG\_STATUS VI\_ATTR\_VXI\_VME\_INTR\_STATUS

# VI\_ATTR\_WIN\_ACCESS

### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range	Default
Read Only Local	ViUInt16	VI_NMAPPED (1) VI_USE_OPERS (2) VI_DEREF_ADDR (3)	VI_NMAPPED

VI\_ATTR\_WIN\_ACCESS specifies the modes in which the current window may be accessed.

- If VI\_NMAPPED, the window is not currently mapped.
- If VI\_USE\_OPERS, the window is accessible through the viPeek*XX*() and viPoke*XX*() operations only.
- If VI\_DEREF\_ADDR, you can either use operations or directly dereference the mapped address as a pointer.

### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_WIN\_ACCESS\_PRIV VI\_ATTR\_WIN\_BASE\_ADDR/VI\_ATTR\_WIN\_BASE\_ADDR\_32/VI\_ATTR VI\_ATTR\_WIN\_BYTE\_ORDER VI\_ATTR\_WIN\_SIZE/VI\_ATTR\_WIN\_SIZE\_32/VI\_ATTR\_WIN\_SIZE\_64 viMapAddress/viMapAddressEx viPeek8/viPeek16/viPeek32/viPeek64

viPoke8/viPoke16/viPoke32/viPoke64

# VI\_ATTR\_WIN\_ACCESS\_PRIV

### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_DATA_PRIV (0)	VI_DATA_PRIV
		VI_DATA_NPRIV	
		(1)	
		VI_PROG_PRIV	
		(2)	
		VI_PROG_NPRIV	
		(3)	
		VI_BLCK_PRIV	
		(4)	
		VI_BLCK_NPRIV	
		(5)	

VI\_ATTR\_WIN\_ACCESS\_PRIV specifies the address modifier to be used in low-level access operations, such as viMapAddress(), viPeek*XX*(), and viPoke*XX*(), when accessing the mapped window.

This attribute is Read/Write when the corresponding session is not mapped (that is, when VI\_ATTR\_WIN\_ACCESS is VI\_NMAPPED. When the session is mapped, this attribute is Read Only.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_ACCESS\_PRIV VI\_ATTR\_SRC\_ACCESS\_PRIV VI\_ATTR\_WIN\_ACCESS VI\_ATTR\_WIN\_BASE\_ADDR/VI\_ATTR\_WIN\_BASE\_ADDR\_32/VI\_ATTR VI\_ATTR\_WIN\_BYTE\_ORDER VI\_ATTR\_WIN\_BYTE\_ORDER

# VI\_ATTR\_WIN\_BASE\_ADDR/VI\_ATTR\_WIN\_BASE

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege Data Type

#### Range

ReadVI\_ATTR\_WIN\_BASE\_ADDR:VI\_ATTR\_WIN\_BASE\_AOnlyViBusAddressOh to FFFFFFFh for 3<br/>applications

VI\_ATTR\_WIN\_BASE\_ADDR\_32: ViUInt32

Oh to FFFFFFFFFFFFF 64-bit applications

VI\_ATTR\_WIN\_BASE\_ADDR\_64: ViUInt64

> VI\_ATTR\_WIN\_BASE\_AD Oh to FFFFFFh

#### VI\_ATTR\_WIN\_BASE\_AD

Oh to FFFFFFFFFFFF

VI\_ATTR\_WIN\_BASE\_ADDR, VI\_ATTR\_WIN\_BASE\_ADDR\_32, and VI\_ATTR\_WIN\_BASE\_ADDR\_64 specify the base address of the interface bus to which this window is mapped. If the value of VI\_ATTR\_WIN\_ACCESS is VI\_NMAPPED, the value of this attribute is meaningless.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_WIN\_ACCESS VI\_ATTR\_WIN\_ACCESS\_PRIV VI\_ATTR\_WIN\_BYTE\_ORDER VI\_ATTR\_WIN\_SIZE/VI\_ATTR\_WIN\_SIZE\_32/VI\_ATTR\_WIN\_SIZE\_64

# VI\_ATTR\_WIN\_BYTE\_ORDER

### **Resource Classes**

GPIB-VXI INSTR, GPIB-VXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege	Data Type	Range		Default	
Read/Write Local	ViUInt16	VI_BIG_] VI_LITTI	ENDIAN (0) LE_ENDIAN (1)	VI_BIG_	_ENDIAN

VI\_ATTR\_WIN\_BYTE\_ORDER specifies the byte order to be used in lowlevel access operations, such as viMapAddress(), viPeek*XX*(), and viPoke*XX*(), when accessing the mapped window.

This attribute is Read/Write when the corresponding session is not mapped (that is, when VI\_ATTR\_WIN\_ACCESS is VI\_NMAPPED. When the session is mapped, this attribute is Read Only.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_BYTE\_ORDER VI\_ATTR\_SRC\_BYTE\_ORDER VI\_ATTR\_WIN\_ACCESS VI\_ATTR\_WIN\_ACCESS\_PRIV VI\_ATTR\_WIN\_BASE\_ADDR/VI\_ATTR\_WIN\_BASE\_ADDR\_32/VI\_ATTR VI\_ATTR\_WIN\_SIZE/VI\_ATTR\_WIN\_SIZE\_32/VI\_ATTR\_WIN\_SIZE\_64

# VI\_ATTR\_WIN\_SIZE/VI\_ATTR\_WIN\_SIZE\_32/VI\_A

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

Access Privilege Data Type

Read VI\_ATTR\_WIN\_SIZE: Only ViBusSize Local

> VI\_ATTR\_WIN\_SIZE\_32: ViUInt32

> > ViUInt64

### Range

#### Default

VI\_ATTR\_WIN\_SIZE: N/A Oh to FFFFFFFh for 32-bit applications

ViUInt32 Oh to FFFFFFFFFFFFFF VI\_ATTR\_WIN\_SIZE\_64: for 64-bit applications

> VI\_ATTR\_WIN\_SIZE\_32: Oh to FFFFFFFh

> VI\_ATTR\_WIN\_SIZE\_64: Oh to FFFFFFFFFFFFFFF

VI\_ATTR\_WIN\_SIZE, VI\_ATTR\_WIN\_SIZE\_32, and VI\_ATTR\_WIN\_SIZE\_64 specify the size of the region mapped to this window. If the value of VI\_ATTR\_WIN\_ACCESS is VI\_NMAPPED, the value of this attribute is meaningless.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_WIN\_ACCESS VI\_ATTR\_WIN\_ACCESS\_PRIV VI\_ATTR\_WIN\_BASE\_ADDR/VI\_ATTR\_WIN\_BASE\_ADDR\_32/VI\_ATTR VI\_ATTR\_WIN\_BYTE\_ORDER

# VI\_ATTR\_WR\_BUF\_OPER\_MODE

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read/Write Local	ViUInt16	VI_FLUSH_ON_ACCESS (1)	VI_FLUSH_WHEN_FULL
		VI_FLUSH_WHEN_FULL	
		(2)	

VI\_ATTR\_WR\_BUF\_OPER\_MODE specifies the operational mode of the formatted I/O write buffer. When the operational mode is set to VI\_FLUSH\_WHEN\_FULL (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up. If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the write buffer is flushed under the same conditions, and also every time a viPrintf() (or related) operation completes.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_RD\_BUF\_OPER\_MODE viFlush viPrintf

# VI\_ATTR\_WR\_BUF\_SIZE

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

Access Privilege	Data Type	Range	Default
Read Only Local	ViUInt32	N/A	N/A
This is the current size of the formatted I/O output buffer for this session. The user can modify this value by calling viSetBuf().

#### **Related Topics**

VI\_ATTR\_RD\_BUF\_SIZE VI\_ATTR\_WR\_BUF\_OPER\_MODE viSetBuf

# **Events**

These topics describe the VISA events. The event descriptions are listed in alphabetical order for easy reference.

Each event description contains a list below the title indicating the supported resource classes, such as GPIB, Serial, etc. The event description contains a brief description of the event attributes. <u>Attributes</u> contains more detailed descriptions of the event attributes.

# VI\_EVENT\_ASRL\_BREAK

Note 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.

## **Resource Classes**

Serial INSTR

Notification that a break signal was received.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_BREAK\_STATE VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_ASRL\_CHAR

Note 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.

## **Resource Classes**

Serial INSTR

Notification that at least one data byte has been received. Each data character will not necessarily result in an event notification. In other words, if multiple data bytes arrive at once, you may get only one event. After receiving this event, you should query the serial port for the number of bytes available via the VI\_ATTR\_ASRL\_AVAIL\_NUM attribute.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_AVAIL\_NUM VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_ASRL\_CTS

Note 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.

## **Resource Classes**

Serial INSTR

Notification that the Clear To Send (CTS) line changed state. If the CTS line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_CTS\_STATE VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_ASRL\_DCD

Note 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.

## **Resource Classes**

Serial INSTR

Notification that the Data Carrier Detect (DCD) line changed state. If the DCD line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_DCD\_STATE VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_ASRL\_DSR

Note 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.

## **Resource Classes**

Serial INSTR

Notification that the Data Set Ready (DSR) line changed state. If the DSR line changes state quickly several times in succession, not all line state changes will necessarily result in event notifications.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_DSR\_STATE VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_ASRL\_RI

Note 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.

## **Resource Classes**

Serial INSTR

Notification that the Ring Indicator (RI) input signal was asserted.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_RI\_STATE VI\_ATTR\_EVENT\_TYPE

## VI\_EVENT\_ASRL\_TERMCHAR

Note 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.

## **Resource Classes**

Serial INSTR

Notification that the termination character has been received. 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

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

#### **Related Topics**

INSTR Resource VI\_ATTR\_ASRL\_AVAIL\_NUM VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_TERMCHAR

## VI\_EVENT\_CLEAR

## **Resource Classes**

GPIB INTFC, VXI SERVANT

Notification that the local controller has been sent a device clear message.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

### **Related Topics**

INTFC Resource SERVANT Resource VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_EXCEPTION

## **Resource Classes**

All I/O session types

This event notifies the application that an error condition has occurred during an operation invocation. In VISA, exceptions are defined as events. The exception-handling model follows the event-handling model for callbacks, and is like any other event in VISA, except that the queueing and suspended handler mechanisms are not allowed.

A VISA operation generating an exception blocks until the exception handler execution is completed. However, an exception handler sometimes may prefer to terminate the program prematurely without returning the control to the operation generating the exception. VISA does not preclude an application from using a platform-specific or language-specific exception handling mechanism from within the VISA exception handler. For example, the C++ try/catch block can be used in an application in conjunction with the C++ throw mechanism from within the VISA exception handler.

When using the C++ try/catch/throw or other exception-handling mechanisms, the control will not return to the VISA system. This has some important repercussions:

- If multiple handlers were installed on the exception event, the handlers that were not invoked prior to the current handler will not be invoked for the current exception.
- The exception context will not be deleted by the VISA system when a C++ exception is used. In this case, the application should delete the exception context as soon as the application has no more use for the context, before terminating the session. An application should use the viClose() operation to delete the exception context.

One situation in which an exception event will not be generated is in the case of asynchronous operations. If the error is detected after the operation is posted—once the asynchronous portion has begun—the status is returned normally via the I/O completion event. However, if an error occurs before the asynchronous portion begins—the error is returned from the asynchronous operation itself—then the exception event will still be raised. This deviation is due to the fact that asynchronous operations already raise an event when they complete, and this I/O completion event may occur in the context of a separate

thread previously unknown to the application. In summary, a single application event handler can easily handle error conditions arising from both exception events and failed asynchronous operations.
Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event. This attribute always has the value of VI_EVENT_EXCEPTION for this event type.
VI_ATTR_STATUS	Contains the status code returned by the operation generating the error.
VI_ATTR_OPER_NAME	Contains the name of the operation generating the event.

### **Related Topics**

VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_OPER\_NAME VI\_ATTR\_STATUS viEnableEvent

# VI\_EVENT\_GPIB\_CIC

**Resource Classes** 

**GPIB INTFC** 

Notification that the GPIB controller has gained or lost CIC (controller in charge) status.

#### Symbolic Name

VI\_ATTR\_EVENT\_TYPE

### Description

Unique logical identifier of the event.

VI\_ATTR\_GPIB\_RECV\_CIC\_STATE Specifies whether the CIC status was gained or lost.

#### **Related Topics**

INTFC Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_GPIB\_RECV\_CIC\_STATE

# VI\_EVENT\_GPIB\_LISTEN

## **Resource Classes**

**GPIB INTFC** 

Notification that the GPIB controller has been addressed to listen.

### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

## **Related Topics**

INTFC Resource VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_GPIB\_TALK

**Resource Classes** 

**GPIB INTFC** 

Notification that the GPIB controller has been addressed to talk.

### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

## **Related Topics**

INTFC Resource VI\_ATTR\_EVENT\_TYPE

# VI\_EVENT\_IO\_COMPLETION

#### **Resources Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI MEMACC, VXI SERVANT

This event notifies the application that an asynchronous operation has completed.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event. This attribute always has the value of VI_EVENT_IO_COMPLETION for this event type.
VI_ATTR_STATUS	Contains the return code of the asynchronous I/O operation that has completed.
VI_ATTR_JOB_ID	Contains the job ID of the asynchronous operation that has completed.
VI_ATTR_BUFFER	Contains the address of the buffer that was used in the asynchronous operation.
VI_ATTR_RET_COUNT/ VI_ATTR_RET_COUNT_32/ VI_ATTR_RET_COUNT_64	Contains the actual number of elements that were asynchronously transferred.
VI_ATTR_OPER_NAME	Contains the name of the operation generating the event.

## **Related Topics**

**INSTR Resource** 

**INTFC Resource** 

MEMACC Resource

SERVANT Resource

SOCKET Resource

VI\_ATTR\_BUFFER VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_JOB\_ID VI\_ATTR\_OPER\_NAME VI\_ATTR\_RET\_COUNT/VI\_ATTR\_RET\_COUNT\_32/VI\_ATTR\_RET\_COU VI\_ATTR\_STATUS

# VI\_EVENT\_PXI\_INTR

## **Resource Classes**

**PXI INSTR** 

This event notifies that a PXI interrupt has occurred.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.
VI_ATTR_PXI_RECV_INTR_SEQ	The index of the interrupt sequence that detected the interrupt condition.
VI_ATTR_PXI_RECV_INTR_DATA	The first PXI/PCI register that was read in the successful interrupt detection sequence.

#### **Related Topics**

INSTR Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_PXI\_RECV\_INTR\_DATA VI\_ATTR\_PXI\_RECV\_INTR\_SEQ

# VI\_EVENT\_SERVICE\_REQ

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, TCPIP INSTR, USB INSTR, VXI INSTR

This event notifies the application that a service request was received from the device or interface associated with the given session.



**Note** When you receive a VI\_EVENT\_SERVICE\_REQ on an instrument session, you must call viReadSTB() to guarantee delivery of future service request events on the given session.

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event. This attribute always has the value of VI\_EVENT\_SERVICE\_REQ for this event type.

#### **Related Topics**

INSTR Resource INTFC Resource VI\_ATTR\_EVENT\_TYPE viReadSTB

# VI\_EVENT\_TRIG

## **Resource Classes**

GPIB INTFC, VXI INSTR, VXI BACKPLANE, VXI SERVANT

This event notifies the application that a trigger interrupt was received from the device. This may be either a hardware or software trigger, depending on the interface and the current session settings.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event. This attribute always has the value of VI_EVENT_TRIG for this event type.
VI ATTR RECV TRIC ID	The identifier of the

VI\_ATTR\_RECV\_TRIG\_ID The identifier of the triggering mechanism on which the specified trigger event was received.

### **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource SERVANT Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_RECV\_TRIG\_ID VI\_ATTR\_TRIG\_ID

# VI\_EVENT\_USB\_INTR

## **Resource Classes**

USB INSTR, USB RAW

This event notifies that a USB interrupt has occurred.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.
VI_ATTR STATUS	Contains the status code returned by this event.
VI_ATTR_USB_RECV_INTR_SIZE	The number of bytes of USB interrupt data that is stored.
VI_ATTR_USB_RECV_INTR_DATA	The actual received data from

the USB Interrupt.

#### **Related Topics**

INSTR Resource RAW Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_STATUS VI\_ATTR\_USB\_RECV\_INTR\_DATA VI\_ATTR\_USB\_RECV\_INTR\_SIZE

# VI\_EVENT\_VXI\_SIGP

**Resource Classes** 

**VXI INSTR** 

This event notifies the application that a VXIbus signal or VXIbus interrupt was received from the device associated with the given session.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event. This attribute always has the value of VI_EVENT_VXI_SIGP for this event type.

VI\_ATTR\_SIGP\_STATUS\_ID The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.

### **Related Topics**

INSTR Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_SIGP\_STATUS\_ID

# VI\_EVENT\_VXI\_VME\_INTR

# **Resource Classes**

**VXI INSTR** 

This event notifies the application that a VXIbus interrupt was received from the device associated with the given session.

Symbolic Name	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event. This attribute always has the value of VI_EVENT_VXI_VME_INTR for this event type.
VI_ATTR_INTR_STATUS_ID	The 32-bit Status/ID value retrieved during the IACK cycle.
VI_ATTR_RECV_INTR_LEVEL	The VXI interrupt level on which the interrupt was received.

### **Related Topics**

INSTR Resource VI\_ATTR\_EVENT\_TYPE VI\_ATTR\_INTR\_STATUS\_ID VI\_ATTR\_RECV\_INTR\_LEVEL

# VI\_EVENT\_VXI\_VME\_SYSFAIL

## **Resource Classes**

VXI BACKPLANE

Notification that the VXI/VME SYSFAIL\* line has been asserted.

### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

## **Related Topics**

BACKPLANE Resource

# VI\_EVENT\_VXI\_VME\_SYSRESET

## **Resource Classes**

VXI BACKPLANE, VXI SERVANT
# Description

Notification that the VXI/VME SYSRESET\* line has been asserted.

## **Event Attributes**

#### Symbolic Name Description

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event

## **Related Topics**

BACKPLANE Resource SERVANT Resource VI\_ATTR\_EVENT\_TYPE

# Operations

These topics describe the VISA operations. The operation descriptions are listed in alphabetical order for easy reference.

Each event description contains a brief *Purpose* statement below the title. You will then see the operation defined in both ANSI C and Visual Basic version 4 syntax, with the parameters set in **boldface** type. A list indicating the supported resource classes, such as GPIB, Serial, etc. is followed by a table that describes each parameter and indicates whether it is an input or output parameter (or both, in some cases). The *Return Values* section describes the completion and error codes, followed by a detailed *Description* section. The *Related Items* section directs you toward related operations, attributes, events, or resource descriptions. If you want to know specifically about attributes, events, and operations of the INSTR Resource, for example, you should navigate to the <u>INSTR</u> <u>Resource</u> topic.

# viAssertIntrSignal

# Purpose

Asserts the specified interrupt or signal.

# C Syntax

ViStatus viAssertIntrSignal(ViSession vi, ViInt16 mode, ViUInt32 statusID)

# Visual Basic Syntax

viAssertIntrSignal&(ByVal **vi**&, ByVal **mode**%, ByVal **statusID**&)

# **Resource Classes**

GPIB-VXI BACKPLANE, VXI BACKPLANE, VXI SERVANT

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mode	IN	This specifies how to assert the interrupt. Refer to the Description section for actual values.
statusID	IN	This is the status value to be presented during an interrupt acknowledge cycle.

# **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INTR_PENDING	An interrupt is still pending from a previous call.
VI_ERROR_INV_MODE	The value specified by the <b>mode</b> parameter is invalid.
VI_ERROR_NSUP_INTR	The interface cannot generate an interrupt on the requested level or with the requested statusID value.

The specified <b>mode</b> is
not supported by this
VISA implementation.

## Description

This operation can be used to assert a device interrupt condition. In VXI, for example, this can be done with either a VXI signal or a VXI interrupt. On certain bus types, the statusID parameter may be ignored. The following table lists the valid values for the **mode** parameter.

Mode	Action Description
VI_ASSERT_USE_ASSIGNED	Use whatever notification method that has been assigned to the local device.
VI_ASSERT_SIGNAL	Send the notification via a VXI signal.
VI_ASSERT_IRQ1 - VI_ASSERT_IRQ7	Send the interrupt via the specified VXI/VME IRQ line. This uses the standard VXI/VME ROAK (Release On AcKnowledge) interrupt mechanism, rather than the older VME RORA (Release On Register Access) mechanism.

#### **Related Topics**

BACKPLANE Resource SERVANT Resource viAssertUtilSignal

# viAssertTrigger

# Purpose

Asserts software or hardware trigger.

# C Syntax

ViStatus viAssertTrigger(ViSession vi, ViUInt16 protocol)

# Visual Basic Syntax

viAssertTrigger&(ByVal vi&, ByVal protocol%)

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI BACKPLANE, PXI INSTR, PXI BACKPLANE, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI BACKPLANE

#### Parameters

Direction	Description
IN	Unique logical identifier to a session.
IN	Trigger protocol to use during assertion. Valid values are:
	Range
	GPIB, Serial, TCPIP, USB
	VI_TRIG_PROT_DEFAULT (0)
	VXI, GPIB-VXI
	VI_TRIG_PROT_DEFAULT (0), VI_TRIG_PROT_ON (1), VI_TRIG_PROT_OFF (2), and VI_TRIG_PROT_SYNC (5)
	PXI
	VI_TRIG_PROT_RESERVE (6) VI_TRIG_PROT_UNRESERVE (7)
	IN IN

#### **Return Values**

#### **Completion Codes** Description

VI\_SUCCESS

The specified trigger was successfully asserted to the device.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT The given session reference is invalid.

VI\_ERROR\_NSUP\_OPER

The given **vi** does not support this operation.

VI\_ERROR\_RSRC\_LOCKED Specified

operation could not be performed because the resource identified by **vi** has been locked for this kind of access.

VI\_ERROR\_INV\_PROT The protocol

The protocol specified is invalid.

VI\_ERROR\_TMO Timeout expired before operation completed.

VI\_ERROR\_RAW\_WR\_PROT\_VIOL Violation of raw write protocol occurred during transfer.

VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw
	read protocol
	occurred during
	transfer.

VI_ERROR_INP_PROT_VIOL	Device reported
	an input protocol

# VI\_ERROR\_BERR

Bus error occurred during transfer.

error during transfer.

VI\_ERROR\_LINE\_IN\_USE The specified trigger line is currently in use.

#### VI\_ERROR\_NCIC

The interface associated with the given **vi** is not currently the controller in charge.

## VI\_ERROR\_NLISTENERS No-listeners

condition is detected (both NRFD and NDAC are unasserted).

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent

state).

VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.

# Description

The viAssertTrigger() operation sources a software or hardware trigger dependent on the interface type.

## Software Triggers for 488.2 Instruments (GPIB, VXI, TCPIP, and USB)

This operation sends an IEEE-488.2 software trigger to the addressed device. For software triggers, VI\_TRIG\_PROT\_DEFAULT is the only valid protocol. The bus-specific details are:

- For a GPIB device, VISA addresses the device to listen and then sends the GPIB GET command.
- For a VXI device, VISA sends the Word Serial Trigger command.
- For a USB device, VISA sends the TRIGGER message ID on the Bulk-OUT pipe.

# Software Triggers for Non-488.2 Instruments (Serial INSTR, TCPIP SOCKET, and USB RAW)

If VI\_ATTR\_IO\_PROT is VI\_PROT\_4882\_STRS, this operations sends "\*TRG\n" to the device; otherwise, this operation is not valid. For software triggers, VI\_TRIG\_PROT\_DEFAULT is the only valid protocol.

## Hardware Triggering for VXI

For hardware triggers to VXI instruments, VI\_ATTR\_TRIG\_ID must first be set to the desired trigger line to use; this operation performs the specified trigger operation on the previously selected trigger line. For VXI hardware triggers, VI\_TRIG\_PROT\_DEFAULT is equivalent to VI\_TRIG\_PROT\_SYNC.

#### **Trigger Reservation for PXI**

For PXI instruments, this operation reserves or releases (unreserves) a trigger line for use in external triggering. For PXI triggers, VI\_TRIG\_PROT\_RESERVE and VI\_TRIG\_PROT\_UNRESERVE are the only valid protocols.

## **Related Topics**

BACKPLANE Resource INSTR Resource INTFC Resource SOCKET Resource

# viAssertUtilSignal

# Purpose

Asserts or deasserts the specified utility bus signal.

# C Syntax

viStatus viAssertUtilSignal(ViSession **vi**, ViUInt16 **line**)

# Visual Basic Syntax

viAssertUtilSignal& (ByVal **vi**&, ByVal **line**%)

# **Resource Classes**

GPIB-VXI BACKPLANE, VXI BACKPLANE, VXI SERVANT

## **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
line	IN	Specifies the utility bus signal to assert. This can be the value VI_UTIL_ASSERT_SYSRESET, VI_UTIL_ASSERT_SYSFAIL, or VI_UTIL_DEASSERT_SYSFAIL.

# **Return Values**

Completion Codes Desc	cription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	<ul> <li>Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.</li> </ul>
VI_ERROR_TMO	Timeout expired before operation completedS.
VI_ERROR_INV_LINE	The value specified by the <b>line</b> parameter is invalid.

# Description

This operation can be used to assert either the SYSFAIL or SYSRESET utility bus interrupts on the VXIbus backplane. This operation is valid only on BACKPLANE (mainframe) and VXI SERVANT (servant) sessions.

Asserting SYSRESET (also known as HARD RESET in the VXI specification) should be used only when it is necessary to promptly terminate operation of all devices in a VXIbus system. This is a serious action that always affects the entire VXIbus system.

#### **Related Topics**

BACKPLANE Resource SERVANT Resource viAssertIntrSignal

# viBufRead

# Purpose

Reads data from device or interface through the use of a formatted I/O read buffer.

# C Syntax

ViStatus viBufRead(ViSession **vi**, ViPBuf **buf**, ViUInt32 **count**, ViPUInt32 **retCount**)

# **Visual Basic Syntax**

viBufRead&(ByVal **vi**&, ByVal **buf**\$, ByVal **count**&, **retCount**&)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	OUT	Location of a buffer to receive data from device.
count	IN	Number of bytes to be read.
retCount	OUT	Number of bytes actually transferred.

#### **Return Values**

# **Completion Codes**

VI\_SUCCESS

#### Description

The operation completed successfully and the END indicator was received (for interfaces that have END indicators). This completion code is returned regardless of whether the termination character is received or the number of bytes read is equal to **count**.

#### VI\_SUCCESS\_TERM\_CHAR

The specified termination character was read but no END indicator was received. This completion code is returned regardless of whether the number of bytes read is equal to **count**.

VI\_SUCCESS\_MAX\_CNT

The number of bytes read is equal to **count**. No END indicator was received and no termination character was read.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT

The given session reference is invalid.

VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_IO	An unknown I/O error occurred during transfer.

# Description

The viBufRead() operation is similar to viRead() and does not perform any kind of data formatting. It differs from viRead() in that the data is read from the formatted I/O read buffer—the same buffer used by viScanf() and related operations—rather than directly from the device. You can intermix this operation with viScanf(), but you should not mix it with viRead().

VI\_NULL is a special value for the **retCount** parameter. If you pass VI\_NULL for **retCount**, the number of bytes transferred is not returned. You may find this useful if you need to know only whether the operation succeeded or failed.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource

SOCKET Resource

<u>viBufWrite</u>

<u>viRead</u>
# viBufWrite

# Purpose

Writes data to a formatted I/O write buffer synchronously.

# C Syntax

ViStatus viBufWrite(ViSession **vi**, ViBuf **buf**, ViUInt32 **count**, ViPUInt32 **retCount**)

# **Visual Basic Syntax**

viBufWrite&(ByVal **vi**&, ByVal **buf**\$, ByVal **count**&, **retCount**&)

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Location of a block of data.
count	IN	Number of bytes to be written.
retCount	OUT	Number of bytes actually transferred.

# **Return Values**

Completion Codes Desc	cription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

# Description

The viBufWrite() operation is similar to viWrite() and does not perform any kind of data formatting. It differs from viWrite() in that the data is written to the formatted I/O write buffer—the same buffer used by viPrintf() and related operations—rather than directly to the device. You can intermix this operation with viPrintf(), but you should not mix it with viWrite().

If this operation returns VI\_ERROR\_TMO, the write buffer for the specified session is cleared.

VI\_NULL is a special value for the **retCount** parameter. If you pass VI\_NULL for **retCount**, the number of bytes transferred is not returned. You may find this useful if you need to know only whether the operation succeeded or failed.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viBufRead viWrite

# viClear

# Purpose

Clears a device.

# C Syntax

ViStatus viClear(ViSession vi)

# Visual Basic Syntax

viClear&(ByVal **vi**&)

## **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.

## **Return Values**

values	
Completion Codes Description	ı
VI_SUCCESS Operat	ion completed ccessfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.

VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.

#### VI\_ERROR\_NLISTENERS

No-listeners condition is detected (both NRFD and NDAC are unasserted).

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

#### VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.

# Description

The viClear() operation clears the device input and output buffers. The bus-specific details are:

## Clear for 488.2 Instruments (GPIB, VXI, TCPIP, and USB)

- For a GPIB device, VISA sends the Selected Device Clear command.
- For a VXI device, VISA sends the Word Serial Clear command.
- For a USB device, VISA sends the INITIATE\_CLEAR and CHECK\_CLEAR\_STATUS commands on the control pipe.

# Clear for Non-488.2 Instruments (Serial INSTR, TCPIP SOCKET, and USB RAW)

- For Serial INSTR sessions, VISA flushes (discards) the I/O output buffer, sends a break, and then flushes (discards) the I/O input buffer.
- For TCPIP SOCKET sessions, VISA flushes (discards) the I/O buffers.
- For USB RAW sessions, VISA resets the endpoints referred to by the attributes VI\_ATTR\_USB\_BULK\_IN\_PIPE and VI\_ATTR\_USB\_BULK\_OUT\_PIPE.

Invoking viClear() also discards the read and write buffers used by the formatted I/O services for that session.

#### **Related Topics**

INSTR Resource

SOCKET Resource

# viClose

# Purpose

Closes the specified session, event, or find list.

# C Syntax

ViStatus viClose(ViObject vi)

# Visual Basic Syntax

viClose&(ByVal **vi**&)

# **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session, event, or find list.

## **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Session closed successfully.
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.
Error Codes	Description
VI_ERROR_INV_OBJEC	T The given object reference is invalid.
VI_ERROR_CLOSING_FAI	LED Unable to deallocate the previously allocated data structures corresponding to this

session or object reference.

# Description

The viClose() operation closes a session, event, or a find list. In this process all the data structures that had been allocated for the specified **vi** are freed. Calling viClose() on a VISA Resource Manager session will also close all I/O sessions associated with that resource manager session.

#### **Related Topics**

viFindRsrc viOpen viOpenDefaultRM VISA Resource Template viWaitOnEvent

# viDisableEvent

## Purpose

Disables notification of the specified event type(s) via the specified mechanism(s).

# C Syntax

ViStatus viDisableEvent(ViSession **vi**, ViEventType **eventType**, ViUInt16 **mechanism**)

# **Visual Basic Syntax**

viDisableEvent&(ByVal vi&, ByVal eventType&, ByVal mechanism%)

# **Resource Classes**

All I/O session types

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
mechanism	IN	Specifies event handling mechanisms to be disabled. The queuing mechanism is disabled by specifying VI_QUEUE (1), and the callback mechanism is disabled by specifying VI_HNDLR (2) or VI_SUSPEND_HNDLR (4). It is possible to disable both mechanisms simultaneously by specifying VI_ALL_MECH (FFFFh).

## **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Event disabled successfully.
VI_SUCCESS_EVENT_DIS	Specified event is already disabled for at least one of the specified mechanisms.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_INV_EVENT	Specified <b>eventType</b> is not supported by the resource.
VI_ERROR_INV_MECH	Invalid <b>mechanism</b> specified.

# Description

The viDisableEvent() operation disables servicing of an event identified by the **eventType** parameter for the mechanisms specified in the **mechanism** parameter. This operation prevents *new* event occurrences from being added to the queue(s). However, event occurrences already existing in the queue(s) are not flushed. Use viDiscardEvents() if you want to discard events remaining in the queue(s).

Specifying VI\_ALL\_ENABLED\_EVENTS for the **eventType** parameter allows a session to stop receiving all events. The session can stop receiving queued events by specifying VI\_QUEUE. Applications can stop receiving callback events by specifying either VI\_HNDLR or VI\_SUSPEND\_HNDLR. Specifying VI\_ALL\_MECH disables both the queuing and callback mechanisms.

Note Calling viDisableEvent() prevents future events from being raised on the given session. When the method returns to the application, it is possible that a callback may still be active, such as on another thread. It is valid for a user to call viDisableEvent() from within a callback, but this is not recommended.

## **Related Topics**

viEnableEvent VISA Resource Template viUninstallHandler

# viDiscardEvents

## Purpose

Discards event occurrences for specified event types and mechanisms in a session.

# C Syntax

ViStatus viDiscardEvents(ViSession **vi**, ViEventType **eventType**, ViUInt16 **mechanism**)

# **Visual Basic Syntax**

viDiscardEvents&(ByVal vi&, ByVal eventType&, ByVal mechanism%)

# **Resource Classes**

All I/O session types

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
mechanism	IN	Specifies the mechanisms for which the events are to be discarded. The VI_QUEUE (1) value is specified for the queuing mechanism and the VI_SUSPEND_HNDLR (4) value is specified for the pending events in the callback mechanism. It is possible to specify both mechanisms simultaneously by specifying VI_ALL_MECH (FFFFh).

**Return Values** 

Completion Codes	Description
VI_SUCCESS	Event queue flushed successfully.
VI_SUCCESS_QUEUE_EMPTY	Operation completed successfully, but queue was already empty.
Error Codes Des	cription

VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_INV_EVENT	Specified <b>eventType</b> is not supported by the resource.
VI_ERROR_INV_MECH	Invalid <b>mechanism</b> specified.

# Description

The viDiscardEvents() operation discards all pending occurrences of the specified event types and mechanisms from the specified session. Specifying VI\_ALL\_ENABLED\_EVENTS for the **eventType** parameter discards events of every type that is enabled for the given session. The information about all the event occurrences which have not yet been handled is discarded. This operation is useful to remove event occurrences that an application no longer needs. The discarded event occurrences are not available to a session at a later time. This operation does not apply to event contexts that have already been delivered to the application.

#### **Related Topics**

viDisableEvent viEnableEvent VISA Resource Template viWaitOnEvent
# viEnableEvent

## Purpose

Enables notification of a specified event.

## C Syntax

ViStatus viEnableEvent(ViSession **vi**, ViEventType **eventType**, ViUInt16 **mechanism**, ViEventFilter **context**)

# Visual Basic Syntax

viEnableEvent&(ByVal **vi**&, ByVal **eventType**&, ByVal **mechanism**%, ByVal **context**&)

### **Resource Classes**

All I/O session types

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
mechanism	IN	Specifies event handling mechanisms to be enabled. The queuing mechanism is enabled by specifying VI_QUEUE (1), and the callback mechanism is enabled by specifying VI_HNDLR (2) or VI_SUSPEND_HNDLR (4).
context	IN	VI_NULL <b>(0)</b> .

#### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Event enabled successfully.
VI_SUCCESS_EVENT_EN	Specified event is already enabled for at least one of the specified mechanisms.
Error Codes	Description
VI_ERROR_INV_OBJ	IECT The given session reference is invalid.
VI_ERROR_INV_EV	ENT Specified eventType is not supported by the resource.
VI_ERROR_INV_ME	ECH Invalid <b>mechanism</b> specified for the event.
VI_ERROR_INV_CON	TEXT Specified event context is invalid.
VI_ERROR_INV_SE	TUP Unable to start

Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent

state).

VI\_ERROR\_HNDLR\_NINSTALLED A handler is not currently installed for the specified event. The session cannot be enabled for the VI\_HNDLR mode of the callback mechanism.

#### VI\_ERROR\_NSUP\_MECH

The specified mechanism is not supported for the given eventType.

## Description

The viEnableEvent() operation enables notification of an event identified by the **eventType** parameter for mechanisms specified in the **mechanism** parameter. The specified session can be enabled to queue events by specifying VI\_QUEUE. Applications can enable the session to invoke a callback function to execute the handler by specifying VI\_HNDLR. The applications are required to install at least one handler to be enabled for this mode. Specifying VI\_SUSPEND\_HNDLR enables the session to receive callbacks, but the invocation of the handler is deferred to a later time. Successive calls to this operation replace the old callback mechanism with the new callback mechanism.

Specifying VI\_ALL\_ENABLED\_EVENTS for the **eventType** parameter refers to all events which have previously been enabled on this session, making it easier to switch between the two callback mechanisms for multiple events.

NI-VISA does not support enabling both the queue and the handler for the same event type on the same session. If you need to use both mechanisms for the same event type, you should open multiple sessions to the resource.

#### **Related Topics**

<u>Events</u> viDisableEvent viEventHandler viInstallHandler VISA Resource Template viUninstallHandler viWaitOnEvent

# viEventHandler

## Purpose

Event service handler procedure prototype.

#### C Syntax

ViStatus \_VI\_FUNCH viEventHandler(ViSession **vi**, ViEventType **eventType**, ViEvent **context**, ViAddr **userHandle**)

# Visual Basic Syntax

N/A

### **Resource Classes**

All I/O session types

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
context	IN	A handle specifying the unique occurrence of an event.
userHandle	IN	A value specified by an application that can be used for identifying handlers uniquely in a session for an event.

#### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Event handled successfully.
VI_SUCCESS_NCHAIN	Event handled successfully. Do not invoke any other handlers on this session for this event.

## Description

viEventHandler() is not an actual VISA operation. Rather, it is the prototype for a user event handler that is installed with the viInstallHandler() operation. The user handler is called whenever a session receives an event and is enabled for handling events in the VI\_HNDLR mode. The handler services the event and returns VI\_SUCCESS on completion. The VISA system automatically invokes the viClose() operation on the event context when a user handler returns.

Because the event context must still be valid after the user handler returns (so that VISA can free it up), an application should not invoke the viClose() operation on an event context passed to a user handler.



**Note** For advanced users—If the user handler will not return to VISA, the application should call viClose() on the event context to manually delete the event object. This situation may occur when a handler throws a C++ exception in response to a VISA exception event.

Normally, an application should always return VI\_SUCCESS from all callback handlers. If a specific handler does not want other handlers to be invoked for the given event for the given session, it should return VI\_SUCCESS\_NCHAIN. No return value from a handler on one session will affect callbacks on other sessions. Future versions of VISA (or specific implementations of VISA) may take actions based on other return values, so a user should return VI\_SUCCESS from handlers unless there is a specific reason to do otherwise.

#### **Related Topics**

vilnstallHandler VISA Resource Template viUninstallHandler

# viFindNext

#### Purpose

Returns the next resource from the list of resources found during a previous call to viFindRsrc().

# C Syntax

ViStatus viFindNext(ViFindList **findList**, ViChar **instrDesc[]**)

## Visual Basic Syntax

viFindNext&(ByVal **findList**&, ByVal **instrDesc**\$)

### **Resource Classes**

VISA Resource Manager

#### Parameters

Name	Direction	Description
findList	IN	Describes a find list. This parameter must be created by viFindRsrc().
instrDesc	OUT	Returns a string identifying the location of a device. Strings can then be passed to viOpen() to establish a session to the given device.

#### **Return Values**

Completion Codes De	scription
VI_SUCCESS	Resource(s) found.
Error Codes	Description
VI_ERROR_INV_OBJEC	T The given object reference is invalid.
VI_ERROR_NSUP_OPEF	The given <b>findList</b> does not support this operation.
VI_ERROR_RSRC_NFOUN	ND There are no more matches.

## Description

The viFindNext() operation returns the next device found in the list created by viFindRsrc(). The list is referenced by the handle that was returned by viFindRsrc().



**Note** The size of the **instrDesc** parameter should be *at least* 256 bytes.

**Related Topics** 

<u>viFindRsrc</u>

VISA Resource Template

# viFindRsrc

#### Purpose

Queries a VISA system to locate the resources associated with a specified interface.

### C Syntax

ViStatus viFindRsrc(ViSession **sesn**, ViString **expr**, ViPFindList **findList**, ViPUInt32 **retcnt**, ViChar **instrDesc[]**)

## **Visual Basic Syntax**

viFindRsrc&(ByVal **sesn**&, ByVal **expr**\$, **findList**&, **retcnt**&, ByVal **instrDesc**\$)

### **Resource Classes**

VISA Resource Manager

#### Parameters

Name	Direction	Description
sesn	IN	Resource Manager session (should always be the session returned from viOpenDefaultRM()).
expr	IN	This is a regular expression followed by an optional logical expression. Refer to the discussion of the Description String in the <i>Description</i> section of this operation.
findList	OUT	Returns a handle identifying this search session. This handle will be used as an input in viFindNext().
retcnt	OUT	Number of matches.
instrDesc	OUT	Returns a string identifying the location of a device. Strings can then be passed to viOpen() to establish a session to the given device.

#### **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Resource(s) found.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>sesn</b> does not support this operation. This operation is supported only by a Resource Manager session.
VI_ERROR_INV_EXPR	Invalid expression specified for search.
VI_ERROR_RSRC_NFOUND	Specified expression does not match any devices.

## Description

The viFindRsrc() operation matches the value specified in the **expr** parameter with the resources available for a particular interface. A regular expression is a string consisting of ordinary characters as well as special characters. You use a regular expression to specify patterns to match in a given string; in other words, it is a search criterion. The viFindRsrc() operation uses a case-insensitive compare feature when matching resource names against the regular expression specified in **expr**. For example, calling viFindRsrc() with "VXI?\*INSTR" would return the same resources as invoking it with "vxi?\*instr".

On successful completion, this function returns the first resource found (**instrDesc**) and returns a count (**retcnt**) to indicate if there were more resources found for the designated interface. This function also returns, in the **findList** parameter, a handle to a find list. This handle points to the list of resources and it must be used as an input to viFindNext(). When this handle is no longer needed, it should be passed to viClose(). Notice that **retcnt** and **findList** are optional parameters. This is useful if only the first match is important, and the number of matches is not needed. If you specify VI\_NULL in the **findList** parameter and the operation completes successfully, VISA automatically invokes viClose() on the find list handle rather than returning it to the application.

- Note The size of the instrDesc parameter should be at least 256 bytes.
- **Note** All resource strings returned by viFindRsrc() will always be recognized by viOpen(). However, viFindRsrc() will not necessarily return all strings that you can pass to viParseRsrc() or viOpen(). This is especially true for network and TCPIP resources. If a resource does not appear in the list, you can explicitly add it in the NI-VISA configuration utility (MAX on Windows, visaconf on UNIX), and then viFindRsrc() will return it. The configuration utility also has other options that expand or limit the set of resources that viFindRsrc() returns.

The search criteria specified in the **expr** parameter has two parts: a regular expression over a resource string, and an optional logical expression over attribute values. The regular expression is matched against the resource strings of resources known to the VISA Resource

Manager. If the resource string matches the regular expression, the attribute values of the resource are then matched against the expression over attribute values. If the match is successful, the resource has met the search criteria and gets added to the list of resources found.

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.
[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.

<b>Regular Expression</b>	Sample Matches
GPIB?*INSTR	Matches GPIB0::2::INSTR, GPIB1::1::1::INSTR, and GPIB-VXI1::8::INSTR.
GPIB[0-9]*::?*INSTR	Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not GPIB-VXI1::8::INSTR.
GPIB[^0]::?*INSTR	Matches GPIB1::1::1::INSTR but not GPIB0::2::INSTR or GPIB12::8::INSTR.
VXI?*INSTR	Matches VXI0::1::INSTR but not GPIB-VXI0::1::INSTR.
GPIB-VXI?*INSTR	Matches GPIB- VXI0::1::INSTR but not VXI0::1::INSTR.
?*VXI[0-9]*::?*INSTR	Matches VXI0::1::INSTR and GPIB-VXI0::1::INSTR.
ASRL[0-9]*::?*INSTR	Matches ASRL1::INSTR but not VXI0::5::INSTR.
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.
?*INSTR	Matches all INSTR (device) resources.
?*VXI[0-9]*::? *MEMACC	Matches VXI0::MEMACC and GPIB-VXI1::MEMACC.
VXI0::?*	Matches VXI0::1::INSTR, VXI0::2::INSTR, and VXI0::MEMACC.
?*	Matches all resources.
visa://hostname/?*	Matches all resources on the specified remote system. The hostname can be represented as either an IP address (dot-notation) or network machine name. This remote system need not be a configured remote system.
/?*	Matches all resources on the local machine. Configured remote systems are not queried.
visa:/ASRL?*INSTR	Matches all ASRL resources on the local machine and returns them in URL format (for example, visa:/ASRL1::INSTR).

You can use the NI-VISA configuration utility (MAX on Windows, visaconf

on UNIX) to access certain NI-VISA servers by default. All expressions without the preceding "/" will be matched with resources on the configured remote systems.

By using the optional attribute expression, you can construct flexible and powerful expressions with the use of logical ANDs (&&), ORs(||), and NOTs (!). You can use equal (==) and unequal (!=) comparators to compare attributes of any type, and other inequality comparators (>, <, >=, <=) to compare attributes of numeric type. Use only global attributes in the attribute expression. Local attributes are not allowed in the logical expression part of the **expr** parameter.

#### **Expr Parameter**

#### Meaning

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

## **Related Topics**

viClose viFindNext VISA Resource Template

# viFlush

#### Purpose

Manually flushes the specified buffers associated with formatted I/O operations and/or serial communication.

# C Syntax

ViStatus viFlush(ViSession vi, ViUInt16 mask)
# Visual Basic Syntax

viFlush&(ByVal **vi**&, ByVal **mask**%)

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR, VXI SERVANT

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mask	IN	Specifies the action to be taken with flushing the buffer. Refer to the <i>Description</i> section for more information.

## **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Buff	ers flushed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_IO	Could not perform Read/Write operation because of I/O error.
VI_ERROR_TMO	The Read/Write operation was aborted because timeout expired while operation was in progress.
VI_ERROR_INV_MASK	The specified <b>mask</b> does not specify a valid flush operation on Read/Write resource.

## Description

The value of **mask** can be one of the following flags.

Flag	Interpretation
VI_READ_BUF (1)	Discard the read buffer contents. If data was present in the read buffer and no END-indicator was present, read from the device until encountering an END indicator (which causes the loss of data). This action resynchronizes the next viScanf() call to read a <terminated RESPONSE MESSAGE&gt;. (Refer to the IEEE 488.2 standard.)</terminated 
VI_READ_BUF_DISCARD (4)	Discard the read buffer contents (does not perform any I/O to the device).
VI_WRITE_BUF (2)	Flush the write buffer by writing all buffered data to the device.
VI_WRITE_BUF_DISCARD (8)	Discard the write buffer contents (does not perform any I/O to the device).
VI_IO_IN_BUF (16)	Discard the low-level I/O receive buffer contents (same as VI_IO_IN_BUF_DISCARD).
VI_IO_IN_BUF_DISCARD	Discard the low-level I/O

(64)	receive buffer contents (does not perform any I/O to the device).
VI_IO_OUT_BUF (32)	Flush the low-level I/O transmit buffer by writing all buffered data to the device.
VI_IO_OUT_BUF_DISCARD (128)	Discard the low-level I/O transmit buffer contents (does not perform any I/O to the device).

It is possible to combine any of these read flags and write flags for different buffers by ORing the flags. However, combining two flags for the same buffer in the same call to viFlush() is illegal.

Notice that when using formatted I/O operations with a session to a Serial device or Ethernet socket, a flush of the formatted I/O buffers also causes the corresponding I/O communication buffers to be flushed. For example, calling viFlush() with VI\_WRITE\_BUF also flushes the VI\_IO\_OUT\_BUF.

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.

#### Implicit versus Explicit Flushing

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.

Flushing a write buffer immediately sends any queued data to the device. The write buffer is maintained by the viPrintf() operation. To explicitly flush the write buffer, you can make a call to the viFlush() operation with a write flag set. In addition, the write buffer is flushed automatically under the following conditions:

- 1. When an END-indicator character is sent (that is, the \n character is specified in the formatting string).
- 2. When the buffer is full.

3. In response to a call to viSetBuf() with the VI\_WRITE\_BUF flag set.

Flushing a read buffer discards the data in the read buffer. This guarantees that the next call to a viScanf() (or related) operation reads data directly from the device rather than from queued data residing in the read buffer. The read buffer is maintained by the viScanf() operation. To explicitly flush the read buffer, you can make a call to the viFlush() operation with a read flag set.

Also, the formatted I/O buffers of a session to a given device are reset whenever that device is cleared. Invoking the viClear() operation will flush the read buffer and discard the contents of the write buffers.

#### **Related Topics**

Automatically Flushing the Formatted I/O Buffers Controlling the Serial 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 INSTR Resource INTFC Resource Manually Flushing the Formatted I/O Buffers Recommendations for Using the VISA Buffers SERVANT Resource SOCKET Resource

<u>viSetBuf</u>

# viGetAttribute

## Purpose

Retrieves the state of an attribute.

## C Syntax

ViStatus viGetAttribute(ViObject **vi**, ViAttr **attribute**, void \* **attrState**)

## **Visual Basic Syntax**

viGetAttribute&(ByVal **vi**&, ByVal **attribute**&, **attrState** as Any)

## **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session, event, or find list.
attribute	IN	Resource attribute for which the state query is made.
attrState	OUT	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual object.

## **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Attribute retrieved successfully.
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given object reference is invalid.
VI_ERROR_NSUP_AT	TR The specified attribute is not defined by the referenced object.

## Description

The viGetAttribute() operation is used to retrieve the state of an attribute for the specified session, event, or find list.

The output parameter **attrState** is of the type of the attribute actually being retrieved. For example, when retrieving an attribute that is defined as a ViBoolean, your application should pass a reference to a variable of type ViBoolean. Similarly, if the attribute is defined as being ViUInt32, your application should pass a reference to a variable of type ViUInt32.

#### **Related Topics**

Attributes

VISA Resource Template

<u>viSetAttribute</u>

# viGpibCommand

## Purpose

Write GPIB command bytes on the bus.

## C Syntax

ViStatus viGpibCommand (ViSession **vi**, ViBuf **buf**, ViUInt32 **count**, ViPUInt32 **retCount**)

## **Visual Basic Syntax**

viGpibCommand&(ByVal vi&, ByVal buf\$, ByVal count&, retCount&)

## **Resource Classes**

**GPIB INTFC** 

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Buffer containing valid GPIB commands.
count	IN	Number of bytes to be written.
retCount	OUT	Number of bytes actually transferred.

## **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD

and NDAC are deasserted).

### VI\_ERROR\_IO

An unknown I/O error occurred during transfer.

## Description

This operation attempts to write **count** number of bytes of GPIB commands to the interface bus specified by **vi**. This operation is valid only on GPIB INTFC (interface) sessions. This operation returns only when the transfer terminates.

If you pass VI\_NULL as the **retCount** parameter to the viGpibCommand() operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed. The command bytes contained in **buf** should be valid IEEE 488-defined Multiline Interface Messages.

#### **Related Topics**

**INTFC Resource** 

# viGpibControlATN

## Purpose

Specifies the state of the ATN line and the local active controller state.

# C Syntax

ViStatus viGpibControlATN(ViSession vi, ViUInt16 mode)

## Visual Basic Syntax

viGpibControlATN& (ByVal vi&, ByVal mode%)

## **Resource Classes**

**GPIB INTFC** 

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mode	IN	Specifies the state of the ATN line and optionally the local active controller state. See the <i>Description</i> section for actual values.

## **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.
VI_ERROR_INV_MODE	The value specified by the <b>mode</b> parameter is invalid.
VI_ERROR_NSUP_MODE	The specified <b>mode</b> is not supported by this VISA implementation.

### Description

This operation asserts or deasserts the GPIB ATN interface line according to the specified mode. The mode can also specify whether the local interface should acquire or release Controller Active status. This operation is valid only on GPIB INTFC (interface) sessions. The following table lists valid values for the **mode** parameter.

Mode	Action Description
VI_GPIB_ATN_DEASSERT	Deassert ATN line. The GPIB interface corresponding to the VISA session goes to standby.
VI_GPIB_ATN_ASSERT	Assert ATN line and take control synchronously without corrupting transferred data. If a data handshake is in progress, ATN is not asserted until the handshake is complete.
VI_GPIB_ATN_DEASSERT_HANDSHAKE	Deassert ATN line, and enter shadow handshake mode. The

local board participates in data handshakes as an Acceptor without actually reading the data. The GPIB interface corresponding to the VISA session goes to standby. VI GPIB ATN ASSERT IMMEDIATE Assert ATN line and take control asynchronously and immediately without regard for any data transfer currently in progress. Generally, this should be used only under error conditions.

It is generally not necessary to use the viGpibControlATN() operation in most applications. Other operations such as viGpibCommand() and viGpibPassControl() modify the ATN and/or CIC state automatically.

#### **Related Topics**

INTFC Resource

viGpibControlREN

# viGpibControlREN

### Purpose

Controls the state of the GPIB Remote Enable (REN) interface line, and optionally the remote/local state of the device.

# C Syntax

ViStatus viGpibControlREN(ViSession vi, ViUInt16 mode)

## Visual Basic Syntax

viGpibControlREN&(ByVal vi&, ByVal mode%)

## **Resource Classes**

GPIB INSTR, GPIB INTFC, USB INSTR

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mode	IN	Specifies the state of the REN line and optionally the device remote/local state. See the <i>Description</i> section for actual values.

## **Return Values**

Completion Codes Description	
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKEE	<ul> <li>Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.</li> </ul>
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_NLISTENERS	No-listeners condition is detected (both NRFD and NDAC are unasserted).
VI_ERROR_NSYS_CNTLR	The interface associated with this session is not the system controller.
VI_ERROR_INV_MODE	The value specified by the <b>mode</b> parameter is
invalid.

## Description

The viGpibControlREN() operation asserts or unasserts the GPIB REN interface line according to the specified mode. The mode can also specify whether the device associated with this session should be placed in local state (before deasserting REN) or remote state (after asserting REN). This operation is valid only if the GPIB interface associated with the session specified by **vi** is currently the system controller.

The following table lists special values for the **mode** parameter.

Value	Description
VI_GPIB_REN_DEASSERT	Deassert REN line.
VI_GPIB_REN_ASSERT	Assert REN line.
VI_GPIB_REN_DEASSERT_GTL	Send the Go To Local (GTL) command and deassert REN line.
VI_GPIB_REN_ASSERT_ADDRESS	Assert REN line and address device.
VI_GPIB_REN_ASSERT_LLO	Send LLO to any devices that are addressed to listen.
VI_GPIB_REN_ASSERT_ADDRESS_LLO	Address this device and send it LLO, putting it in RWLS.

#### VI\_GPIB\_REN\_ASSERT\_GTL

Send the Go To Local command (GTL) to this device.

### **Related Topics**

INSTR Resource INTFC Resource viGpibControlATN

# viGpibPassControl

### Purpose

Tell the GPIB device at the specified address to become controller in charge (CIC).

# C Syntax

ViStatus viGpibPassControl(ViSession **vi**, ViUInt16 **primAddr**, ViUInt16 **secAddr**)

# **Visual Basic Syntax**

viGPIBPassControl& (ByVal vi&, ByVal primAddr%, ByValsec Addr%)

### **Resource Classes**

**GPIB INTFC** 

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
primAddr	IN	Primary address of the GPIB device to which you want to pass control.
secAddr	IN	Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value VI_NO_SEC_ADDR.

# **Return Values**

Completion Codes Desc	cription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKE	D Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.
VI_ERROR_NLISTENERS	5 No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error

occurred during transfer.

VI\_ERROR\_INV\_PARAMETER The primary or secondary address is invalid.

# Description

This operation passes controller in charge status to the device indicated by **primAddr** and **secAddr**, and then deasserts the ATN line. This operation assumes that the targeted device has controller capability. This operation is valid only on GPIB INTFC (interface) sessions.

#### **Related Topics**

**INTFC Resource** 

# viGpibSendIFC

# Purpose

Pulse the interface clear line (IFC) for at least 100 microseconds.

# C Syntax

ViStatus viGpibSendIFC (ViSession vi)

# Visual Basic Syntax

viGpibSendIFC& (ByVal **vi**&)

### **Resource Classes**

**GPIB INTFC** 

### Parameters

Name	Direction Description	
vi	IN	Unique logical identifier to a session.

# **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_NSYS_CNTLR	The interface associated with this session is not the system controller.

### Description

This operation asserts the IFC line and becomes controller in charge (CIC). The local board must be the system controller. This operation is valid only on GPIB INTFC (interface) sessions.

### **Related Topics**

**INTFC Resource** 

# viln8/viln16/viln32/viln64, viln8Ex/viln16Ex/viln32Ex/viln64Ex

### Purpose

Reads in an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified memory space and offset.

# C Syntax

ViStatus viIn8(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViPUInt8 **val8**)

ViStatus viIn16(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViPUInt16 **val16**)

ViStatus viIn32(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViPUInt32 **val32**)

ViStatus viIn64(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViPUInt64 **val64**)

ViStatus viIn8Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViPUInt8 **val8**)

ViStatus viIn16Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViPUInt16 **val16**)

ViStatus viIn32Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViPUInt32 **val32**)

ViStatus viIn64Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViPUInt64 **val64**)

# **Visual Basic Syntax**

viIn8&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, **val8** as Byte) viIn16&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, **val16**%) viIn32&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, **val32**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
space	IN	Specifies the address space. Refer to the table included in the <i>Description</i> section for more information.
offset	IN	Offset (in bytes) of the address or register from which to read. For viInXX() operations, this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viInXXEx() operations, this is always a 64-bit value.
<b>val8</b> , <b>val16</b> , <b>val32</b> or <b>val64</b>	OUT	Data read from bus (8 bits for viIn8[Ex](), 16 bits for viIn16[Ex] (), 32 bits for viIn32[Ex](), and 64 bits for viIn64[Ex]()).

#### **Return Values**

С

ompletion Codes	Description
VI_SUCCESS	Operation completed successfully.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT

VI\_ERROR\_NSUP\_OPER

The given session reference is invalid.

The given **vi** does not support this operation.

#### VI\_ERROR\_RSRC\_LOCKED

Specified operation could not be performed because the resource identified by **vi** has been locked for this kind of access.

VI\_ERROR\_BERR Bus error

occurred during transfer.

VI\_ERROR\_INV\_SPACE

VI\_ERROR\_INV\_OFFSET

VI\_ERROR\_NSUP\_OFFSET

Invalid address **space** specified.

Invalid **offset** specified.

Specified **offset** is not accessible from this hardware.

#### VI\_ERROR\_NSUP\_WIDTH

Specified width is not supported by this hardware.

#### VI\_ERROR\_NSUP\_ALIGN\_OFFSET

The specified offset is not properly aligned for the access width of the operation.

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

## Description

The viIn*XX*[Ex]() operations use the specified address space to read in 8, 16, 32, or 64 bits of data, respectively, from the specified **offset**. These operations do not require viMapAddress() to be called prior to their invocation.

The following table lists the valid entries for specifying address **space**.

Value	Description
VXI, VME, and	VI_A16_SPACE (1)
GPIB-VXI	VI_A24_SPACE (2)
	VI_A32_SPACE (3)
	VI_A64_SPACE (4)
PXI INSTR	VI_PXI_CFG_SPACE (10) VI_PXI_BAR0_SPACE (11) to VI_PXI_BAR5_SPACE (16)
PXI MEMACC	VI_PXI_ALLOC_SPACE (9)
FireWire INSTR	VI_FIREWIRE_DFLT_SPACE (5)

# **INSTR Specific**

Notice that the **offset** parameter to these operations for an INSTR Resource is the offset address relative to the device's allocated address base for the corresponding address space that was specified. For example, if **space** specifies VI\_A16\_SPACE, then **offset** specifies the offset from the logical address base address of the specified VXI device. If **space** specifies VI\_A24\_SPACE or VI\_A32\_SPACE, then **offset** specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

### **MEMACC Specific**

For a MEMACC Resource, the **offset** parameter specifies an absolute address.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET viOut8/viOut16/viOut32/viOut64, viOut8Ex/viOut16Ex/viOut32Ex/viOut64Ex

# vilnstallHandler

# Purpose

Installs handlers for event callbacks.

# C Syntax

ViStatus viInstallHandler(ViSession **vi**, ViEventType **eventType**, ViHndlr **handler**, ViAddr **userHandle**)

# Visual Basic Syntax

N/A

# **Resource Classes**

All I/O session types

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
handler	IN	Interpreted as a valid reference to a handler to be installed by a client application.
userHandle	IN	A value specified by an application that can be used for identifying handlers uniquely for an event type.

#### **Return Values**

Completion Codes	Description
------------------	-------------

VI\_SUCCESS

Event handler installed successfully.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT

The given session reference is invalid.

VI\_ERROR\_INV\_EVENT

Specified eventType is not supported by the resource.

VI\_ERROR\_INV\_HNDLR\_REF

The given handler reference is invalid.

VI_ERROR_HNDLR_NINSTALLED	The handler was not installed. This may be returned if an application attempts to install multiple handlers for the same event on the
	same session.

# Description

The viInstallHandler() operation allows applications to install handlers on sessions. The handler specified in the **handler** parameter is installed along with any previously installed handlers for the specified event. Applications can specify a value in the **userHandle** parameter that is passed to the handler on its invocation. VISA identifies handlers uniquely using the handler reference and this value.

VISA allows applications to install multiple handlers for an **eventType** on the same session. You can install multiple handlers through multiple invocations of the viInstallHandler() operation, where each invocation adds to the previous list of handlers. If more than one handler is installed for an **eventType**, each of the handlers is invoked on every occurrence of the specified event(s). VISA specifies that the handlers are invoked in Last In First Out (LIFO) order.

#### **Related Topics**

<u>viEnableEvent</u> <u>viEventHandler</u> <u>VISA Resource Template</u> <u>viUninstallHandler</u>

# viLock

# Purpose

Establishes an access mode to the specified resources.
## C Syntax

ViStatus viLock(ViSession **vi**, ViAccessMode **lockType**, ViUInt32 **timeout**, ViKeyId **requestedKey**, ViChar **accesskey[]**)

# Visual Basic Syntax

viLock&(ByVal **vi**&, ByVal **lockType**&, ByVal **timeout**&, ByVal **requestedKey**\$, ByVal **accesskey**\$)

### **Resource Classes**

All I/O session types

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
lockType	IN	Specifies the type of lock requested, either VI_EXCLUSIVE_LOCK (1) or VI_SHARED_LOCK (2).
timeout	IN	Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning an error.
requestedKey	IN	This parameter is not used and should be set to VI_NULL when <b>lockType</b> is VI_EXCLUSIVE_LOCK. Refer to the <i>Description</i> section for more details about using VI_SHARED_LOCK.
accessKey	OUT	This parameter should be set to VI_NULL when <b>lockType</b> is VI_EXCLUSIVE_LOCK. When <b>lockType</b> is VI_SHARED_LOCK, the resource returns a unique access key for the lock if the operation succeeds. This <b>accessKey</b> can then be passed to other

sessions to share the lock.

**Return Values** 

**Completion Codes** 

#### Description

VI\_SUCCESS

Specified access mode was acquired.

VI\_SUCCESS\_NESTED\_EXCLUSIVE Specified access mode is successfully acquired, and this session has nested exclusive locks.

VI\_SUCCESS\_NESTED\_SHARED Specified access mode i

access mode is successfully acquired, and this session has nested shared locks.

Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified <b>lockType</b> cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_INV_LOCK_TYPE	Specified <b>lockType</b> is not supported by

this resource.

#### VI\_ERROR\_INV\_ACCESS\_KEY 7

#### The requestedKey

value passed in is not a valid **accessKey** to the specified resource.

VI\_ERROR\_TMO

Specified **lockType** could not be obtained within the specified **timeout** period.

## Description

This operation is used to obtain a lock on the specified resource. The caller can specify the type of lock requested—exclusive or shared lock—and the length of time the operation will suspend while waiting to acquire the lock before timing out. This operation can also be used for sharing and nesting locks.

The **requestedKey** and the **accessKey** parameters apply only to shared locks. These parameters are not applicable when using the lock type VI\_EXCLUSIVE\_LOCK; in this case, **requestedKey** and **accessKey** should be set to VI\_NULL. VISA allows user applications to specify a key to be used for lock sharing, through the use of the **requestedKey** parameter. Alternatively, a user application can pass VI\_NULL for the **requestedKey** parameter when obtaining a shared lock, in which case VISA will generate a unique access key and return it through the **accessKey** parameter. If a user application does specify a **requestedKey** value, VISA will try to use this value for the **accessKey**. As long as the resource is not locked, VISA will use the **requestedKey** as the access key and grant the lock. When the operation succeeds, the **requestedKey** will be copied into the user buffer referred to by the **accessKey** parameter.

Note If requesting a VI\_SHARED\_LOCK, the size of the accessKey parameter should be at least 256 bytes.

The session that gained a shared lock can pass the **accessKey** to other sessions for the purpose of sharing the lock. The session wanting to join the group of sessions sharing the lock can use the key as an input value to the **requestedKey** parameter. VISA will add the session to the list of sessions sharing the lock, as long as the **requestedKey** value matches the **accessKey** value for the particular resource. The session obtaining a shared lock in this manner will then have the same access privileges as the original session that obtained the lock.

It is also possible to obtain nested locks through this operation. To acquire nested locks, invoke the viLock() operation with the same lock type as the previous invocation of this operation. For each session, viLock() and viUnlock() share a lock count, which is initialized to 0. Each invocation of viLock() for the same session (and for the same **lockType**) increases the lock count. In the case of a shared lock, it returns with the

same **accessKey** every time. When a session locks the resource a multiple number of times, it is necessary to invoke the viUnlock() operation an equal number of times in order to unlock the resource. That is, the lock count increments for each invocation of viLock(), and decrements for each invocation of viUnlock(). A resource is actually unlocked only when the lock count is 0.

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 to the same resource are serviced or returned with a locking error, depending on the operation and the type of lock used. 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.

The locking mechanism works for all processes and resources existing on the same computer. When using remote resources, however, the networking protocol may not provide the ability to pass lock requests to the remote device or resource. In this case, locks will behave as expected from multiple sessions on the same computer, but not necessarily on the remote device. For example, when using the VXI-11 protocol, exclusive lock requests can be sent to a device, but shared locks can only be handled locally.

#### **Related Topics**

VISA Resource Template

<u>viUnlock</u>

# viMapAddress/viMapAddressEx

# Purpose

Maps the specified memory space into the process's address space.

# C Syntax

ViStatus viMapAddress(ViSession **vi**, ViUInt16 **mapSpace**, ViBusAddress **mapBase**, ViBusSize **mapSize**, ViBoolean **access**, ViAddr **suggested**, ViPAddr **address**)

ViStatus viMapAddressEx(ViSession **vi**, ViUInt16 **mapSpace**, ViBusAddress64 **mapBase**, ViBusSize **mapSize**, ViBoolean **access**, ViAddr **suggested**, ViPAddr **address**)

#### **Visual Basic Syntax**

viMapAddress&(ByVal **vi**&, ByVal **mapSpace**%, ByVal **mapBase**&, ByVal **mapSize**&, ByVal **access**%, ByVal **suggested**&, **address**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mapSpace	IN	Specifies the address space to map. Refer to the <i>Description</i> section for more information.
mapBase	IN	Offset (in bytes) of the memory to be mapped. Refer to the <i>Description</i> section for more information. For viMapAddress(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMapAddressEx(), this is always a 64-bit value.
mapSize	IN	Amount of memory to map (in bytes).
access	IN	VI_FALSE (0).
suggested	IN	If <b>suggested</b> parameter is not VI_NULL (0), the operating system attempts to map the memory to the address specified in <b>suggested</b> . There is no guarantee, however, that the memory will be mapped to that address. This operation may map the memory into an address region different from <b>suggested</b> .

address OUT Address in your process space where the memory was mapped.

### **Return Values**

Completion Codes Desc	ription
VI_SUCCESS	Mapping successful.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSE	Invalid offset specified.
VI_ERROR_NSUP_OFFSE	T Specified region is not accessible from this hardware.
VI_ERROR_TMO	viMapAddress() could not acquire resource or perform mapping before the timer expired.
VI_ERROR_INV_SIZE	Invalid size of window specified.
VI_ERROR_ALLOC	Unable to allocate window of at least the requested size.
VI_ERROR_INV_ACC_MO	DE Invalid access mode.
VI_ERROR_WINDOW_MAP	PED The specified session already

contains a mapped window.

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

### Description

The viMapAddress() operation maps in a specified memory space. The memory space that is mapped is dependent on the type of interface specified by the **vi** parameter and the **mapSpace** parameter. The **address** parameter returns the address in your process space where memory is mapped. The following table lists the valid entries for the **mapSpace** parameter.

Value	Description
VXI, VME, and GPIB-VXI	VI_A16_SPACE (1) VI_A24_SPACE (2) VI_A32_SPACE (3) VI_A64_SPACE (4)
PXI INSTR	VI_PXI_CFG_SPACE (10) VI_PXI_BAR0_SPACE (11) to VI_PXI_BAR5_SPACE (16)
PXI MEMACC	VI_PXI_ALLOC_SPACE (9)
FireWire INSTR	VI_FIREWIRE_DFLT_SPACE (5)



**Note** On some hardware platforms, the low-level driver may have limitations on the parameters to this function. For example, on VXI resources **mapBase** should be a multiple of **mapSize** for best results. If these limitations prevent NI-VISA from mapping the full region you request (**mapSize** bytes starting at **mapBase**), the function will return an error such as VI\_ERROR\_NSUP\_OFFSET or VI\_ERROR\_ALLOC.

## **INSTR Specific**

Notice that **mapBase** specified in the viMapAddress() operation for an INSTR Resource is the offset address relative to the device's allocated address base for the corresponding address space that was specified. For example, if **mapSpace** specifies VI\_A16\_SPACE, then **mapBase** specifies the offset from the logical address base address of the specified VXI device. If **mapSpace** specifies VI\_A24\_SPACE or VI\_A32\_SPACE, then **mapBase** specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

## **MEMACC Specific**

For a MEMACC Resource, the **mapBase** parameter specifies an absolute address.



**Note** The output address is not necessarily always a pointer. It may be possible for viMapAddress to succeed and output a token address value of 0. This is not the same as a NULL pointer, even though the value of NULL is 0. Obviously, this situation cannot happen if the address is a pointer that the user can dereference. Regardless, you should determine whether viMapAddress succeeded or failed by checking the returned status, not the output value of the address.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET viUnmapAddress

# viMapTrigger

## Purpose

Map the specified trigger source line to the specified destination line.

# C Syntax

viStatus viMapTrigger(ViSession **vi**, ViInt16 **trigSrc**, ViInt16 **trigDest**, ViUInt16 **mode**)

## **Visual Basic Syntax**

viMapTrigger& (ByVal **vi**&, ByVal **trigSrc**%, ByVal **trigDest**%, ByVal **mode**%)

### **Resource Classes**

GPIB-VXI BACKPLANE, PXI BACKPLANE, VXI BACKPLANE

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
trigSrc	IN	Source line from which to map. Refer to the <i>Description</i> section for actual values.
trigDest	IN	Destination line to which to map. Refer to the <i>Description</i> section for actual values.
mode	IN	VI_NULL

### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Operation completed successfully.
VI_SUCCESS_TRIG_MAPPE	D The path from <b>trigSrc</b> to <b>trigDest</b> is already mapped.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_MODE	The value specified by the <b>mode</b> parameter is invalid.
VI_ERROR_LINE_IN_USE	One of the specified lines ( <b>trigSrc</b> or <b>trigDest</b> ) is currently in use.

VI\_ERROR\_INV\_LINE

One of the specified lines (**trigSrc** or **trigDest**) is invalid.

VI\_ERROR\_NSUP\_LINE

One of the specified lines (**trigSrc** or **trigDest**) is not supported by this VISA implementation.

#### Description

This operation can be used to map one trigger line to another. This operation is valid only on BACKPLANE (mainframe) sessions.

Value	Action Description
VI_TRIG_TTL0 - VI_TRIG_TTL7	Map the specified VXI or PXI TTL trigger line.
VI_TRIG_ECL0 - VI_TRIG_ECL1	Map the specified VXI ECL trigger line.
VI_TRIG_PANEL_IN	Map the controller's front panel trigger input line.
VI_TRIG_PANEL_OUT	Map the controller's front panel trigger output line.

If this operation is called multiple times on the same BACKPLANE Resource with the same source trigger line and different destination trigger lines, the result will be that when the source trigger line is asserted, all of the specified destination trigger lines will also be asserted. If this operation is called multiple times on the same BACKPLANE Resource with different source trigger lines and the same destination trigger line, the result will be that when any of the specified source trigger lines is asserted, the destination trigger line will also be asserted.

Note Mapping a trigger line (as either source or destination) multiple times requires special hardware capabilities and is not guaranteed to be implemented.

Refer to <u>VI\_ATTR\_PXI\_SRC\_TRIG\_BUS</u> or <u>VI\_ATTR\_PXI\_DEST\_TRIG\_BUS</u> for information about how to map a trigger between bus segments in a multisegment PXI chassis.

#### **Related Topics**

BACKPLANE Resource

VI\_ATTR\_PXI\_DEST\_TRIG\_BUS

VI\_ATTR\_PXI\_SRC\_TRIG\_BUS

# viMemAlloc/viMemAllocEx

## Purpose

Allocates memory from a resource's memory region.

# C Syntax

ViStatus viMemAlloc(ViSession **vi**, ViBusSize **size**, ViPBusAddress **offset**) ViStatus viMemAllocEx(ViSession **vi**, ViBusSize **size**, ViPBusAddress64 **offset**)

# Visual Basic Syntax

viMemAlloc&(ByVal **vi**&, ByVal **size**&, **offset**&)

### **Resource Classes**

GPIB-VXI INSTR, PXI MEMACC, VXI INSTR

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
size	IN	Specifies the size of the allocation.
offset	OUT	Returns the offset of the allocated memory. For viMemAlloc(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMemAllocEx(), this is always a 64-bit value.

### **Return Values**

Completion Codes Des	cription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKE	D Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_SIZE	Invalid <b>size</b> specified.
VI_ERROR_ALLOC	Unable to allocatte shared memory block of the requested <b>size</b> .
VI_ERROR_MEM_NSHARE	D The device does not export any memory.

# Description

The memory region referenced by the **offset** returned from viMemAlloc() can be accessed with the high-level operations viMoveInXX() and viMoveOutXX(), or mapped using viMapAddress(). When using viMemAllocEx(), the **offset** returned may be accessed by the viMoveInXXEx() and viMoveOutXXEx() operations, and mapped using viMapAddressEx(). Note that for viMemAllocEx(), the offset could be above the 4 GB boundary. If your device cannot access this memory, you should use viMemAlloc() instead.

#### VXI/GPIB-VXI INSTR Specific

Notice that the **offset** parameter to these operations for an INSTR Resource is the offset address relative to the device's allocated address base. The viMemAlloc() and viMemAllocEx() operations return an **offset** into a device's memory region allocated for use by this session. If the device to which the given **vi** refers is on the local interface card, the memory can be allocated either on the device itself or on the computer's system memory.

#### **PXI/PCI MEMACC Specific**

For a MEMACC Resource, the **offset** parameter specifies an absolute address. This is a physical address in system memory and can be used for device DMA.

#### **Related Topics**

**INSTR Resource** 

viMapAddress/viMapAddressEx

viMemFree/viMemFreeEx

viMoveIn8/viMoveIn16/viMoveIn32/viMoveIn64, viMoveIn8Ex/viMoveIn16Ex/viMoveIn32Ex/viMoveIn64Ex

viMoveOut8/viMoveOut16/viMoveOut32/viMoveOut64, viMoveOut8Ex/viMoveOut16Ex/viMoveOut32Ex/viMoveOut64Ex

# viMemFree/viMemFreeEx

#### Purpose

Frees memory previously allocated using the viMemAlloc() operation.
## C Syntax

ViStatus viMemFree(ViSession **vi**, ViBusAddress **offset**) ViStatus viMemFreeEx(ViSession **vi**, ViBusAddress64 **offset**)

## Visual Basic Syntax

viMemFree&(ByVal vi&, ByVal offset&)

## **Resource Classes**

GPIB-VXI INSTR, PXI MEMACC, VXI INSTR

### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
offset	IN	Specifies the memory previously allocated with viMemAlloc() or viMemAllocEx. For viMemFree(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMemFreeEx(), this is always a 64-bit value.

## **Return Values**

<b>Completion Codes</b>	Descripti	on
VI_SUCCESS	Oper	ation completed successfully.
Error Codes		Description
VI_ERROR_INV_C	BJECT	The given session reference is invalid.
VI_ERROR_NSUP	_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_INV_C	)FFSET	Invalid <b>offset</b> specified.
VI ERROR WINDOW	MAPPED	The specified offset

VI\_ERROR\_WINDOW\_MAPPED The specified **offset** is currently in use by viMapAddress().

## Description

The viMemFree() operation frees the memory previously allocated using viMemAlloc(). The viMemFreeEx() operation frees the memory previously allocated using viMemAllocEx(). If the specified offset has been mapped using viMapAddress() or viMapAddressEx(), it must be unmapped before it can be freed.

### **Related Topics**

INSTR Resource viMapAddress/viMapAddressEx viMemAlloc/viMemAllocEx viUnmapAddress

# viMove/viMoveEx

## Purpose

Moves a block of data.

## C Syntax

ViStatus viMove(ViSession **vi**, ViUInt16 **srcSpace**, ViBusAddress **srcOffset**, ViUInt16 **srcWidth**, ViUInt16 **destSpace**, ViBusAddress **destOffset**, ViUInt16 **destWidth**, ViBusSize **length**)

ViStatus viMoveEx(ViSession vi, ViUInt16 srcSpace, ViBusAddress64 srcOffset, ViUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress64 destOffset, ViUInt16 destWidth, ViBusSize length)

## **Visual Basic Syntax**

viMove&(ByVal **vi**&, ByVal **srcSpace**%, ByVal **srcOffset**&, ByVal **srcWidth**%, ByVal **destSpace**%, ByVal **destOffset**&, ByVal **destWidth**%, ByVal **length**&)

### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, VXI INSTR, VXI MEMACC

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
srcSpace	IN	Specifies the address space of the source.
srcOffset	IN	Offset of the starting address or register from which to read. For viMove(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMoveEx(), this is always a 64-bit value.
srcWidth	IN	Specifies the data width of the source.
destSpace	IN	Specifies the address space of the destination.
destOffset	IN	Offset of the starting address or register to which to write. For viMove(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMoveEx(), this is always a 64-bit value.
destWidth	IN	Specifies the data width of the destination.
length	IN	Number of elements to transfer, where the data width of the elements to transfer is identical to the source data

width.

## **Return Values**

values	
Completion Codes Desc	cription
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given session reference is invalid.
VI_ERROR_NSUP_OPI	ER The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCH	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPAC	CE Invalid source or destination space specified.

VI\_ERROR\_INV\_OFFSET Invalid source or destination offset specified.

VI_ERROR_INV_WIDTH	Invalid source or destination width specified.
VI_ERROR_NSUP_OFFSET	Specified source or destination offset is not accessible from this hardware.
VI_ERROR_NSUP_VAR_WIDTH	Cannot support source and destination widths that are different.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_LENGTH	Invalid length specified.

## Description

The viMove() and viMoveEx() operations move data from the specified source to the specified destination. The source and the destination can either be local memory or the offset of the interface with which this MEMACC Resource is associated. These operations use the specified data width and address space. In some systems, such as VXI, users can specify additional settings for the transfer, such as byte order and access privilege, by manipulating the appropriate attributes.

The following table lists the valid entries for specifying address space.

Value	Description
VI_A16_SPACE (1)	Address the A16 address space of the VXI/MXI bus.
VI_A24_SPACE (2)	Address the A24 address space of the VXI/MXI bus.
VI_A32_SPACE (3)	Address the A32 address space of the VXI/MXI bus.
VI_LOCAL_SPACE (0)	Address process-local memory (using a virtual address).
VI_OPAQUE_SPACE (FFFFh)	Addresses potentially volatile data (using a virtual address).

The following table lists the valid entries for specifying widths.

Value	Description
VI_WIDTH_8 (1)	Performs 8-bit (D08) transfers.
VI_WIDTH_16 (2)	Performs 16-bit (D16) transfers.
VI_WIDTH_32 (4)	Performs 32-bit (D32) transfers.

VI\_WIDTH\_64 Performs 64-bit (D64) transfers. (8)

All VXI and PXI accesses performed by the viMove() and viMoveEx() operations use either the same or successive offsets, depending on the increment value specified by VI\_ATTR\_SRC\_INCREMENT and VI\_ATTR\_DEST\_INCREMENT.

If **srcSpace** is VI\_LOCAL\_SPACE, viMove() will ignore VI\_ATTR\_SRC\_INCREMENT. If **destSpace** is VI\_LOCAL\_SPACE, viMove() will ignore VI\_ATTR\_DEST\_INCREMENT. Local accesses always increment the offset for each index in a multi-element transfer, rather than using the increment specified by the attributes. If **srcSpace** is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, viMove() will honor VI\_ATTR\_SRC\_INCREMENT. If **destSpace** is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, viMove() will honor VI\_ATTR\_DEST\_INCREMENT. While VI\_OPAQUE\_SPACE uses a process-local virtual address, it is not necessarily pointing to system memory, so it may be a FIFO. Therefore, VI\_ATTR\_SRC/DEST\_INCREMENT do indeed apply.

## **INSTR Specific**

If **srcSpace** is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, **srcOffset** is a relative address of the device associated with the given INSTR resource. Similarly, if **destspace** is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, **destOffset** is a relative address of the device associated with the given INSTR resource.

The primary intended use of this operation with an INSTR session is to synchronously move data to or from the device. Therefore, either the **srcSpace** or **destSpace** parameter will usually be VI\_LOCAL\_SPACE.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET or

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

## **MEMACC Specific**

The **destOffset** and **srcOffset** parameters specify absolute addresses. Notice also that the **length** specified in the viMove() and viMoveEx() operations is the number of elements (of the size corresponding to the **srcWidth** parameter) to transfer, beginning at the specified offsets. Therefore, **srcOffset + length\*srcWidth** cannot exceed the total amount of memory exported by the given **srcSpace**. Similarly, **destOffset** + **length\*srcWidth** cannot exceed the total amount of memory exported by the given **destSpace**.

### **Related Topics**

INSTR Resource

MEMACC Resource

VI\_ATTR\_DEST\_INCREMENT

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET

VI\_ATTR\_SRC\_INCREMENT

viMoveAsync/viMoveAsyncEx

# viMoveAsync/viMoveAsyncEx

## Purpose

Moves a block of data asynchronously.

## C Syntax

ViStatus viMoveAsync(ViSession **vi**, ViUInt16 **srcSpace**, ViBusAddress **srcOffset**, ViUInt16 **srcWidth**, ViUInt16 **destSpace**, ViBusAddress **destOffset**, ViUInt16 **destWidth**, ViBusSize **length**, ViPJobId **jobId**)

ViStatus viMoveAsyncEx(ViSession vi, ViUInt16 srcSpace, ViBusAddress64 srcOffset, ViUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress64 destOffset, ViUInt16 destWidth, ViBusSize length, ViPJobId jobId)

## Visual Basic Syntax

N/A

### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, VXI INSTR, VXI MEMACC

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
srcSpace	IN	Specifies the address space of the source.
srcOffset	IN	Offset of the starting address or register from which to read. For viMove(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMoveEx(), this is always a 64-bit value.
srcWidth	IN	Specifies the data width of the source.
destSpace	IN	Specifies the address space of the destination.
destOffset	IN	Offset of the starting address or register to which to write. For viMove(), this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMoveEx(), this is always a 64-bit value.
destWidth	IN	Specifies the data width of the destination.
length	IN	Number of elements to transfer, where the data width of the elements to transfer is identical to the source data

width.

jobid OUT Job identifier of this asynchronous move operation.

## **Return Values**

Completion Codes Des	cription
VI_SUCCESS A	synchronous operation successfully queued.
VI_SUCCESS_SYNC	Operation performed synchronously.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKEI	Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.
VI_ERROR_QUEUE_ERROI	R Unable to queue move operation (usually due to the I/O completion event not being enabled or insufficient space in the session's queue).
VI_ERROR_IN_PROGRESS	Unable to queue the asynchronous operation because there is already an operation in progress.

## Description

The viMoveAsync() and viMoveAsyncEx() operations asynchronously move data from the specified source to the specified destination. This operation queues up the transfer in the system, then it returns immediately without waiting for the transfer to carry out or complete. When the transfer terminates, a VI\_EVENT\_IO\_COMPLETION event is generated, which indicates the status of the transfer.

This operation returns **jobld**, which you can use either with viTerminate() to abort the operation or with VI\_EVENT\_IO\_COMPLETION events to identify which asynchronous move operations completed. VISA will never return VI\_NULL for a valid **jobld**.

The source and the destination can either be local memory or the offset of the interface with which this INSTR or MEMACC Resource is associated. This operation uses the specified data width and address space. In some systems, such as VXI, users can specify additional settings for the transfer, such as byte order and access privilege, by manipulating the appropriate attributes.

The following table lists the valid entries for specifying address space.

Value	Description
VI_A16_SPACE (1)	Address the A16 address space of the VXI/MXI bus.
VI_A24_SPACE (2)	Address the A24 address space of the VXI/MXI bus.
VI_A32_SPACE (3)	Address the A32 address space of the VXI/MXI bus.
VI_LOCAL_SPACE (0)	Address process-local memory (using a virtual address).
VI_OPAQUE_SPACE (FFFFh)	Addresses potentially volatile data (using a virtual address).

The following table lists the valid entries for specifying widths.

Value	Description
VI_WIDTH_8 (1)	Performs 8-bit (D08) transfers.
VI_WIDTH_16 (2)	Performs 16-bit (D16) transfers.
VI_WIDTH_32 (4)	Performs 32-bit (D32) transfers.
VI_WIDTH_64 (8)	Performs 64-bit (D64) transfers.

All VXI and PXI accesses performed by the viMoveAsync() and viMoveAsyncEx() operations use either the same or successive offsets, depending on the increment value specified by VI ATTR SRC INCREMENT and VI ATTR DEST INCREMENT.

If **srcSpace** is VI\_LOCAL\_SPACE, viMove() will ignore VI\_ATTR\_SRC\_INCREMENT. If **destSpace** is VI\_LOCAL\_SPACE, viMove() will ignore VI\_ATTR\_DEST\_INCREMENT. Local accesses always increment the offset for each index in a multi-element transfer, rather than using the increment specified by the attributes. If **srcSpace** is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, viMove() will honor VI\_ATTR\_SRC\_INCREMENT. If **destSpace** is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, viMove() will honor VI\_ATTR\_DEST\_INCREMENT. While VI\_OPAQUE\_SPACE uses a process-local virtual address, it is not necessarily pointing to system memory, so it may be a FIFO. Therefore, VI\_ATTR\_SRC/DEST\_INCREMENT do indeed apply.

## **INSTR Specific**

If **srcSpace** is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, **srcOffset** is a relative address of the device associated with the given INSTR resource. Similarly, if **destspace** is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, **destOffset** is a relative address of the device associated with the given INSTR resource.

The primary intended use of this operation with an INSTR session is to asynchronously move data to or from the device. Therefore, either the **srcSpace** or **destSpace** parameter will usually be VI\_LOCAL\_SPACE.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET or

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

## **MEMACC Specific**

The **destOffset** and **srcOffset** parameters specify absolute addresses. Notice also that the length specified in the viMoveAsync() and viMoveAsyncEx() operations is the number of elements (of the size corresponding to the **srcWidth** parameter) to transfer, beginning at the specified offsets. Therefore, **srcOffset + length\*srcWidth** cannot exceed the total amount of memory exported by the given **srcSpace**. Similarly, **destOffset + length\*srcWidth** cannot exceed the total amount of memory exported by the given **destSpace**.

### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET VI\_ATTR\_SRC\_INCREMENT VI\_EVENT\_IO\_COMPLETION viMove/viMoveEx

# viMoveIn8/viMoveIn16/viMoveIn32/viMoveIn64, viMoveIn8Ex/viMoveIn16Ex/viMoveIn32Ex/viMove

## Purpose

Moves a block of data from the specified address space and offset to local memory.

## C Syntax

ViStatus viMoveIn8(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt8 **buf8**)

ViStatus viMoveIn16(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt16 **buf16**)

ViStatus viMoveIn32(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt32 **buf32**)

ViStatus viMoveIn64(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt64 **buf64**)

ViStatus viMoveIn8Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt8 **buf8**)

ViStatus viMoveIn16Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt16 **buf16**)

ViStatus viMoveIn32Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt32 **buf32**)

ViStatus viMoveIn64Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt64 **buf64**)

## **Visual Basic Syntax**

viMoveIn8&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&, **buf8** as Byte)

viMoveIn16&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&, **buf16**%)

viMoveIn32&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&, **buf32**&)

### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
space	IN	Specifies the address space. Refer to the table included in the <i>Description</i> section.
offset	IN	Offset (in bytes) of the starting address to read. For viMoveInXX() operations, this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viMoveInXXEx() operations, this is always a 64-bit value.
length	IN	Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, 32, or 64 bits).
buf8, buf16, buf32, or buf64	OUT	Data read from bus (8 bits for viMoveIn8[Ex](), 16 bits for viMoveIn16[Ex](), 32 bits for viMoveIn32[Ex](), and 64 bits for viMoveIn64[Ex]()).

## **Return Values**

Varaco	
Completion Codes Des	cription
VI_SUCCESS Operation completed successfully.	
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given session reference is invalid.
VI_ERROR_NSUP_OP	ER The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOC	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPAC	CE Invalid address space specified.
VI_ERROR_INV_OFFS	ET Invalid offset specified.
VI_ERROR_NSUP_OFF	SET Specified offset is

not accessible from this hardware.

VI_ERROR_NSUP_WIDTH	Specified width is
	not supported by this hardware.

#### VI\_ERROR\_INV\_LENGTH

Invalid length specified.

#### VI\_ERROR\_NSUP\_ALIGN\_OFFSET

The specified offset is not properly aligned for the access width of the operation.

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
# Description

The viMoveInXX[Ex]() operations use the specified address space to read in 8, 16, 32, or 64 bits of data, respectively, from the specified offset. These operations do not require viMapAddress() to be called prior to their invocation.

The following table lists the valid entries for specifying address space.

Value	Description
VXI, VME, and	VI_A16_SPACE (1)
GPIB-VXI	VI_A24_SPACE (2)
	VI_A32_SPACE (3)
	VI_A64_SPACE (4)
PXI INSTR	VI_PXI_CFG_SPACE (10)
	VI_PXI_BAR5_SPACE (11) to VI_PXI_BAR5_SPACE (16)
PXI MEMACC	VI_PXI_ALLOC_SPACE (9)
FireWire INSTR	VI_FIREWIRE_DFLT_SPACE (5)

For these operations, VISA ignores the attribute

VI\_ATTR\_DEST\_INCREMENT and increments the local buffer address for each element. It is valid for the VISA driver to copy the data into the user buffer at any width it wants. In other words, even if the width is a byte (8-bit), the VISA driver can perform 32-bit PCI burst accesses because it is just memory, to improve throughput.

# **INSTR Specific**

Notice that the **offset** parameter to these operations for an INSTR Resource is the offset address relative to the device's allocated address base for the corresponding address space that was specified. For example, if **space** specifies VI\_A16\_SPACE, then **offset** specifies the offset from the logical address base address of the specified VXI device. If **space** specifies VI\_A24\_SPACE or VI\_A32\_SPACE, then **offset** specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

Notice also that the **length** specified in the viMoveInXX() operations for an INSTR Resource is the number of elements (of the **size** corresponding to the operation) to transfer, beginning at the specified **offset**. Therefore, **offset + length\*size** cannot exceed the amount of memory exported by the device in the given **space**.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

# **MEMACC Specific**

For a MEMACC Resource, the **offset** parameter specifies an absolute address.

Notice also that the **length** parameter to these operations for a MEMACC Resource is the number of elements (of the **size** corresponding to the operation) to transfer, beginning at the specified **offset**. Therefore, **offset + length\*size** cannot exceed the total amount of memory available in the given **space**.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_FIREWIRE\_SRC\_UPPER\_OFFSET viMoveOut8/viMoveOut16/viMoveOut32/viMoveOut64, viMoveOut8Ex/viMoveOut16Ex/viMoveOut32Ex/viMoveOut64Ex

# viMoveOut8/viMoveOut16/viMoveOut32/viMoveOutviMoveOut8Ex/viMoveOut16Ex/viMoveOut32Ex/v

#### Purpose

Moves a block of data from local memory to the specified address space and offset.

# C Syntax

ViStatus viMoveOut8(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt8 **buf8**)

ViStatus viMoveOut16(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt16 **buf16**)

ViStatus viMoveOut32(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt32 **buf32**)

ViStatus viMoveOut64(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViBusSize **length**, ViAUInt64 **buf64**)

ViStatus viMoveOut8Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt8 **buf8**)

ViStatus viMoveOut16Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt16 **buf16**)

ViStatus viMoveOut32Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt32 **buf32**)

ViStatus viMoveOut64Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViBusSize **length**, ViAUInt64 **buf64**)

# **Visual Basic Syntax**

viMoveOut8&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&,**buf8** as Byte)

viMoveOut16&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&, **buf16**%)

viMoveOut32&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **length**&, **buf32**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
space	IN	Specifies the address space. Refer to the table included in the <i>Description</i> section.
offset	IN	Offset (in bytes) of the device to write to. For viMoveOut <i>XX</i> () operations, this is a 32-bit value for 32-bit applications and a 64- bit value for 64-bit applications. For viMoveOut <i>XX</i> Ex() operations, this is always a 64- bit value.
length	IN	Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, 32, or 64 bits).
buf8, buf16, buf32, or buf64	IN	Data to write to bus (8 bits for viMoveOut8[Ex](), 16 bits for viMoveOut16[Ex](), 32 bits for viMoveOut32[Ex](), and 64 bits for viMoveOut64[Ex]()).

# **Return Values**

Valaco		
Completion Codes Des	cription	
VI_SUCCESS	Operation completed successfully.	
Error Codes	Description	
VI_ERROR_INV_OBJE	CT The given session reference is invalid.	
VI_ERROR_NSUP_OP	ER The given <b>vi</b> does not support this operation.	
VI_ERROR_RSRC_LOC	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.	
VI_ERROR_BERR	Bus error occurred during transfer.	
VI_ERROR_INV_SPA	CE Invalid address space specified.	
VI_ERROR_INV_OFFS	ET Invalid offset specified.	
VI_ERROR_NSUP_OFF	SET Specified offset is	

not accessible from this hardware.

VI_ERROR_NSUP_WIDTH	Specified width is
	not supported by this hardware.

#### VI\_ERROR\_INV\_LENGTH

Invalid length specified.

#### VI\_ERROR\_NSUP\_ALIGN\_OFFSET

The specified offset is not properly aligned for the access width of the operation.

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

# Description

The viMoveOut*XX*[Ex]() operations use the specified address space to write 8, 16, 32, or 64 bits of data, respectively, to the specified offset. These operations do not require viMapAddress() to be called prior to their invocation.

The following table lists the valid entries for specifying address space.

Interface	Values
VXI, VME, and	VI_A16_SPACE (1)
GPIB-VXI	VI_A24_SPACE (2)
	VI_A32_SPACE (3)
	VI_A64_SPACE (4)
PXI INSTR	VI_PXI_CFG_SPACE (10) VI_PXI_BAR0_SPACE (11) to VI_PXI_BAR5_SPACE (16)
PXI MEMACC	VI_PXI_ALLOC_SPACE (9)
FireWire INSTR	VI_FIREWIRE_DFLT_SPACE (5)



For these operations, VISA ignores the attribute

VI\_ATTR\_SRC\_INCREMENT and increments the local buffer address for each element. It is valid for the VISA driver to copy the data out of the user buffer at any width it wants. In other words, even if the width is a byte (8-bit), the VISA driver can perform 32-bit PCI burst accesses because it is just memory, to improve throughput.

# **INSTR Specific**

Notice that the **offset** parameter to these operations for an INSTR Resource is the offset address relative to the device's allocated address base for the corresponding address space that was specified. For example, if **space** specifies VI\_A16\_SPACE, then **offset** specifies the offset from the logical address base address of the specified VXI device. If **space** specifies VI\_A24\_SPACE or VI\_A32\_SPACE, then **offset** specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

Notice also that the **length** specified in the viMoveInXX() operations for an INSTR Resource is the number of elements (of the **size** corresponding to the operation) to transfer, beginning at the specified **offset**. Therefore, **offset + length\*size** cannot exceed the amount of memory exported by the device in the given **space**.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

# **MEMACC Specific**

For a MEMACC Resource, the **offset** parameter specifies an absolute address.

Notice also that the **length** parameter to these operations for a MEMACC Resource is the number of elements (of the **size** corresponding to the operation) to transfer, beginning at the specified **offset**. Therefore, **offset + length\*size** cannot exceed the total amount of memory available in the given **space**.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_DEST\_INCREMENT VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET viMoveIn8/viMoveIn16/viMoveIn32/viMoveIn64, viMoveIn8Ex/viMoveIn16Ex/viMoveIn32Ex/viMoveIn64Ex

# viOpen

# Purpose

Opens a session to the specified resource.

# C Syntax

ViStatus viOpen(ViSession **sesn**, ViRsrc **rsrcName**, ViAccessMode **accessMode**, ViUInt32 **openTimeout**, ViPSession **vi**)

#### **Visual Basic Syntax**

viOpen&(ByVal **sesn**&, ByVal **rsrcName**\$, ByVal **accessMode**&, ByVal **openTimeout**&, **vi**&)

# **Resource Classes**

VISA Resource Manager

#### Parameters

Name	Direction	Description
sesn	IN	Resource Manager session (should always be a session returned from viOpenDefaultRM()).
rsrcName	IN	Unique symbolic name of a resource. Refer to the <i>Description</i> section for more information.
accessMode	IN	Specifies the mode by which the resource is to be accessed. Refer to the <i>Description</i> section for valid values. If the parameter value is VI_NULL, the session uses VISA-supplied default values.
openTimeout	IN	Specifies the maximum time period (in milliseconds) that this operation waits before returning an error. This does not set the I/O timeout – to do that you must call viSetAttribute() with the attribute VI_ATTR_TMO_VALUE.
vi	OUT	Unique logical identifier reference to a session.

**Return Values** 

Completion Codes	Description
VI_SUCCESS	Session opened successfully.
VI_SUCCESS_DEV_NPRESENT	Session opened successfully, but the device at the specified address is not responding.
VI_WARN_CONFIG_NLOADED	The specified configuration either does not exist or could not be loaded; using VISA-specified defaults.

**Error Codes** 

#### Description

VI\_ERROR\_INV\_OBJECT T

VI\_ERROR\_NSUP\_OPER

The given session reference is invalid.

The given **sesn** does not support this operation. This operation is supported only by a Resource Manager session.

VI\_ERROR\_INV\_RSRC\_NAME Invalid resource reference specified. Parsing

error.

VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to open a session.
VI_ERROR_RSRC_BUSY	The resource is valid, but VISA cannot currently access it.
VI_ERROR_RSRC_LOCKED	Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_TMO	A session to the resource could not be obtained within the specified openTimeout period.

VI\_ERROR\_LIBRARY\_NFOUND A code library

required by VISA could not be located or loaded.

VI\_ERROR\_INTF\_NUM\_NCONFIG The interface type is valid, but the specified interface number is not configured.

#### VI\_ERROR\_MACHINE\_NAVAIL The remote machine does not exist or is not accepting any connections. If the NI-VISA server is installed and running on the remote machine, it may have an incompatible version or may be listening on a different port.

VI\_ERROR\_NPERMISSION Access to the remote machine is

denied.

# Description

The viOpen() operation opens a session to the specified resource. It returns a session identifier that can be used to call any other operations of that resource. The address string passed to viOpen() must uniquely identify a resource. Refer to <u>VISA Resource Syntax and Examples</u> for the syntax of resource strings and examples.

For the parameter **accessMode**, the value VI\_EXCLUSIVE\_LOCK (1) is used to acquire an exclusive lock immediately upon opening a session; if a lock cannot be acquired, the session is closed and an error is returned. The value VI\_LOAD\_CONFIG (4) is used to configure attributes to values specified by some external configuration utility. Multiple access modes can be used simultaneously by specifying a *bit-wise OR* of the values other than VI\_NULL. NI-VISA currently supports VI\_LOAD\_CONFIG only on Serial INSTR sessions.

All resource strings returned by viFindRsrc() will always be recognized by viOpen(). However, viFindRsrc() will not necessarily return all strings that you can pass to viParseRsrc() or viOpen(). This is especially true for network and TCPIP resources.

#### **Related Topics**

viClose viFindRsrc viOpenDefaultRM viParseRsrc VISA Resource Manager VISA Resource Template

# viOpenDefaultRM

# Purpose

This function returns a session to the Default Resource Manager resource.

# C Syntax

ViStatus viOpenDefaultRM(ViPSession sesn)

# Visual Basic Syntax

viOpenDefaultRM&(sesn&)

# **Resource Classes**

VISA Resource Manager

#### **Parameters**

Name	Direction Description	
sesn	OUT	Unique logical identifier to a Default Resource Manager session.

#### **Return Values**

# Completion CodesDescriptionVI\_SUCCESSSession to the<br/>Default Resource<br/>Manager resource<br/>created successfully.VI\_WARN\_CONFIG\_NLOADEDAt least one<br/>configured Passport<br/>module could not be<br/>loaded.

#### **Error Codes**

#### Description

VI\_ERROR\_SYSTEM\_ERROR

VI\_ERROR\_ALLOC

Insufficient system resources to create a session to the Default Resource

Manager resource.

The VISA system failed to initialize.

VI\_ERROR\_INV\_SETUP

Some implementationspecific configuration file is corrupt or does not exist.

VI_ERROR_LIBRARY_NFOUND	A code library
	required by VISA
	could not be located
	or loaded.

# Description

The viOpenDefaultRM() function must be called before any VISA operations can be invoked. The first call to this function initializes the VISA system, including the Default Resource Manager resource, and also returns a session to that resource. Subsequent calls to this function return unique sessions to the same Default Resource Manager resource.

When a Resource Manager session is passed to viClose(), not only is that session closed, but also all find lists and device sessions (which that Resource Manager session was used to create) are closed.

#### **Related Topics**

viClose viFindRsrc viOpen VISA Resource Manager VISA Resource Template

# viOut8/viOut16/viOut32/viOut64, viOut8Ex/viOut16Ex/viOut32Ex/viOut64Ex

#### Purpose

Writes an 8-bit, 16-bit, 32-bit, or 64-bit value to the specified memory space and offset.

# C Syntax

ViStatus viOut8(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViUInt8 **val8**)

ViStatus viOut16(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViUInt16 **val16**)

ViStatus viOut32(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViUInt32 **val32**)

ViStatus viOut64(ViSession **vi**, ViUInt16 **space**, ViBusAddress **offset**, ViUInt64 **val64**)

ViStatus viOut8Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViUInt8 **val8**)

ViStatus viOut16Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViUInt16 **val16**)

ViStatus viOut32Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViUInt32 **val32**)

ViStatus viOut64Ex(ViSession **vi**, ViUInt16 **space**, ViBusAddress64 **offset**, ViUInt64 **val64**)

# **Visual Basic Syntax**

viOut8&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **val8** as Byte) viOut16&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **val16**%) viOut32&(ByVal **vi**&, ByVal **space**%, ByVal **offset**&, ByVal **val32**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
space	IN	Specifies the address space. Refer to the table included in the <i>Description</i> section for more information.
offset	IN	Offset (in bytes) of the address or register to which to write. For viOut <i>XX</i> () operations, this is a 32-bit value for 32-bit applications and a 64-bit value for 64-bit applications. For viOut <i>XX</i> Ex() operations, this is always a 64-bit value.
val8, val16, val32, or val64	IN	Data to write to bus (8 bits for viOut8[Ex](), 16 bits for viOut16[Ex](), 32 bits for viOut32[Ex](), and 64 bits for viOut64[Ex]()).

#### **Return Values**

**Completion Codes** 

Description

VI\_SUCCESS

Operation completed successfully.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT

The given session reference is invalid.

VI\_ERROR\_NSUP\_OPER Th

The given **vi** does not support this operation.

VI\_ERROR\_RSRC\_LOCKED Specified

operation could not be performed because the resource identified by **vi** has been locked for this kind of access.

VI\_ERROR\_BERR Bus error occurred during transfer. VI\_ERROR\_INV\_SPACE Invalid address space specified.

VI\_ERROR\_INV\_OFFSET Invalid offset specified.

VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an

inconsistent state).
## Description

The viOut*XX*[Ex]() operations use the specified address space to write 8, 16, 32, or 64 bits of data, respectively, from the specified **offset**. These operations do not require viMapAddress() to be called prior to their invocation.

The following table lists the valid entries for specifying address space.

Value	Description
VXI, VME, and GPIB-VXI	VI_A16_SPACE (1) VI_A24_SPACE (2) VI_A32_SPACE (3) VI_A64_SPACE (4)
PXI INSTR	VI_PXI_CFG_SPACE (10) VI_PXI_BAR0_SPACE (11) to VI_PXI_BAR5_SPACE (16)
PXI MEMACC	VI_PXI_ALLOC_SPACE (9)
FireWire INSTR	VI_FIREWIRE_DFLT_SPACE (5)
Note viOut8 and viO	Out16 do not support FireWire INSTR.



# **INSTR Specific**

Notice that the **offset** parameter to these operations for an INSTR Resource is the offset address relative to the device's allocated address base for the corresponding address space that was specified. For example, if **space** specifies VI\_A16\_SPACE, then **offset** specifies the offset from the logical address base address of the specified VXI device. If **space** specifies VI\_A24\_SPACE or VI\_A32\_SPACE, then **offset** specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.

### **MEMACC Specific**

For a MEMACC Resource, the **offset** parameter specifies an absolute address.

#### **Related Topics**

INSTR Resource MEMACC Resource VI\_ATTR\_FIREWIRE\_DEST\_UPPER\_OFFSET viln8/viln16/viln32/viln64, viln8Ex/viln16Ex/viln32Ex/viln64Ex

# viParseRsrc

## Purpose

Parse a resource string to get the interface information.

# C Syntax

ViStatus viParseRsrc(ViSession **sesn**, ViRsrc **rsrcName**, ViPUInt16 **intfType**, ViPUInt16 **intfNum**)

# **Visual Basic Syntax**

viParseRsrc&(ByVal sesn&, ByVal rsrcName\$, intfType%, intfNum%)

## **Resource Classes**

VISA Resource Manager

#### Parameters

Name	Direction	Description
sesn	IN	Resource Manager session (should always be the Default Resource Manager for VISA returned from viOpenDefaultRM()).
rsrcName	IN	Unique symbolic name of a resource.
intfType	OUT	Interface type of the given resource string.
intfNum	OUT	Board number of the interface of the given resource string.

Completion Codes Descrip	tion
VI_SUCCESS Reso	ource string is valid.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>sesn</b> does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAM	E Invalid resource reference specified. Parsing error.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to parse the string.

VI_ERROR_LIBRARY_NFOUND	A code library
	required by VISA
	could not be
	located or loaded.

VI\_ERROR\_INTF\_NUM\_NCONFIG The interface type is valid, but the specified interface number is not configured.

## Description

This operation parses a resource string to verify its validity. It should succeed for all strings returned by viFindRsrc() and recognized by viOpen(). This operation is useful if you want to know what interface a given resource descriptor would use without actually opening a session to it. Refer to <u>VISA Resource Syntax and Examples</u> for the syntax of resource strings and examples.

The values returned in **intfType** and **intfNum** correspond to the attributes VI\_ATTR\_INTF\_TYPE and VI\_ATTR\_INTF\_NUM. These values would be the same if a user opened that resource with viOpen() and queried the attributes with viGetAttribute().

Calling viParseRsrc() with "VXI::1::INSTR" will produce the same results as invoking it with "vxi::1::instr".

#### **Related Topics**

VI\_ATTR\_INTF\_NUM VI\_ATTR\_INTF\_TYPE viFindRsrc viOpen viParseRsrcEx VISA Resource Template

# viParseRsrcEx

### Purpose

Parse a resource string to get extended interface information.

## C Syntax

ViStatus viParseRsrcEx (ViSession **rmSesn**, ViRsrc **rsrcName**, ViPUInt16 **intfType**, ViPUInt16 **intfNum**, ViChar **rsrcClass[]**, ViChar **expandedUnaliasedName[]**, ViChar **aliasIfExists[]**);

### **Visual Basic Syntax**

viParseRsrcEx& (ByVal **sesn**&, ByVal **desc**\$, **intfType**%, **intfNum**%, ByVal **rsrcClass**\$, ByVal **expandedUnaliasedName**\$, ByVal **aliasIfExists**\$)

## **Resource Classes**

VISA Resource Manager

#### Parameters

Name	Direction	Description
sesn	IN	Resource Manager session (should always be the Default Resource Manager for VISA returned from viOpenDefaultRM()).
RsrcName	IN	Unique symbolic name of a resource.
intfType	OUT	Interface type of the given resource string.
intfNum	OUT	Board number of the interface of the given resource string.
RsrcClass	OUT	Specifies the resource class (for example, "INSTR") of the given resource string.
Expanded UnaliasedName	OUT	This is the expanded version of the given resource string. The format should be similar to the VISA-defined canonical resource name.
AliasIfExists	OUT	Specifies the user-defined alias for the given resource string.

### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Resource string is valid.
VI_WARN_EXT_FUNC_NIMPL	The operation succeeded, but a lower level driver did not implement the extended functionality.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>sesn</b> does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAM	E Invalid resource reference specified. Parsing error.

VI\_ERROR\_RSRC\_NFOUND

Insufficient location information or

resource not present in the system.

#### VI\_ERROR\_ALLOC Insufficient system resources to parse the string.

V	[_ERROF	R_LIBF	RARY_	NFOUND	A code library
					required by VISA
					could not be
					located or loaded.
/Τ	ERROR	INTF	NUM	NCONFIG	The interface type

VI\_ERROR\_INTF\_NUM\_NCONFIG The interface type is valid, but the specified interface number is not configured.

## Description

This operation parses a resource string to verify its validity. It should succeed for all strings returned by viFindRsrc() and recognized by viOpen(). This operation is useful if you want to know what interface a given resource descriptor would use without actually opening a session to it. Refer to <u>VISA Resource Syntax and Examples</u> for the syntax of resource strings and examples.

The values returned in **intfType** and **intfNum** correspond to the attributes VI\_ATTR\_INTF\_TYPE and VI\_ATTR\_INTF\_NUM. These values would be the same if a user opened that resource with viOpen() and queried the attributes with viGetAttribute().

The value returned in **unaliasedExpandedRsrcName** should in most cases be identical to the VISA-defined canonical resource name. However, there may be cases where the canonical name includes information that the driver may not know until the resource has actually been opened. In these cases, the value returned in this parameter must be semantically similar.

The value returned in **aliasIfExists** allows programmatic access to userdefined aliases.

Calling viParseRsrc() with "VXI::1::INSTR" will produce the same results as invoking it with "vxi::1::instr".

#### **Related Topics**

VI\_ATTR\_INTF\_NUM VI\_ATTR\_INTF\_TYPE viFindRsrc viOpen viParseRsrc VISA Resource Template

# viPeek8/viPeek16/viPeek32/viPeek64

### Purpose

Reads an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified address.

# C Syntax

void viPeek8(ViSession **vi**, ViAddr **addr**, ViPUInt8 **val8**) void viPeek16(ViSession **vi**, ViAddr **addr**, ViPUInt16 **val16**) void viPeek32(ViSession **vi**, ViAddr **addr**, ViPUInt32 **val32**) void viPeek64(ViSession **vi**, ViAddr **addr**, ViPUInt64 **val64**)

### **Visual Basic Syntax**

viPeek8(ByVal **vi**&, ByVal **addr**&, **val8** as Byte) viPeek16(ByVal **vi**&, ByVal **addr**&, **val16**%) viPeek32(ByVal **vi**&, ByVal **addr**&, **val32**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
addr	IN	Source address to read the value.
val8, val16, val32, or val64	OUT	Data read from bus (8 bits for viPeek8(), 16 bits for viPeek16(), 32 bits for viPeek32(), and 64 bits for viPeek64()).

# **Return Values**

None

# Description

The viPeek*XX*() operations read an 8-bit, 16-bit, 32-bit value, or 64-bit value, respectively, from the address location specified in addr. The address must be a valid memory address in the current process mapped by a previous viMapAddress() call.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.



Note If you use NI Spy to debug these operations, enable the Force peek/poke calls to appear in NI Spy option in Measurement & Automation Explorer (Windows), visaconf (Linux), or NI-VISA Configuration (Mac OS X). If you do not enable this option, NI Spy might not log these operations.

### **Related Topics**

INSTR Resource MEMACC Resource VI ATTR FIREWIRE WIN UPPER OFFSET

VI\_ATTR\_WIN\_ACCESS

viMapAddress/viMapAddressEx

viPoke8/viPoke16/viPoke32/viPoke64

# viPoke8/viPoke16/viPoke32/viPoke64

### Purpose

Writes an 8-bit, 16-bit, 32-bit, or 64-bit value to the specified address.

# C Syntax

void viPoke8(ViSession **vi**, ViAddr **addr**, ViUInt8 **val8**) void viPoke16(ViSession **vi**, ViAddr **addr**, ViUInt16 **val16**) void viPoke32(ViSession **vi**, ViAddr **addr**, ViUInt32 **val32**) void viPoke64(ViSession **vi**, ViAddr **addr**, ViUInt64 **val64**)

### **Visual Basic Syntax**

viPoke8(ByVal **vi**&, ByVal **addr**&, ByVal **val8** as Byte) viPoke16(ByVal **vi**&, ByVal **addr**&, ByVal **val16**%) viPoke32(ByVal **vi**&, ByVal **addr**&, ByVal **val32**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
addr	IN	Destination address to store the value.
val8, val16, val32, or val64	IN	Value to be stored (8 bits for viPoke8(), 16 bits for viPoke16(), 32 bits for viPoke32(), 64 bits for viPoke64()).

# **Return Values**

None

# Description

The viPoke*XX*() operations store the content of an 8-bit, 16-bit, 32-bit value, or 64-bit value, respectively, to the address pointed to by addr. The address must be a valid memory address in the current process mapped by a previous viMapAddress() call.

To specify the full 48-bit offset for these methods on a FireWire INSTR session, a call to viSetAttribute using

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET is needed prior to calling these methods. Subsequent calls to viSetAttribute are needed only if the upper 16 bits of the FireWire offset change.



Note If you use NI Spy to debug these operations, enable the Force peek/poke calls to appear in NI Spy option in Measurement & Automation Explorer (Windows), visaconf (Linux), or NI-VISA Configuration (Mac OS X). If you do not enable this option, NI Spy might not log these operations.

### **Related Topics**

INSTR Resource MEMACC Resource

VI\_ATTR\_FIREWIRE\_WIN\_UPPER\_OFFSET

VI\_ATTR\_WIN\_ACCESS

viMapAddress/viMapAddressEx

viPeek8/viPeek16/viPeek32/viPeek64

# viPrintf

### Purpose

Converts, formats, and sends the parameters (designated by...) to the device as specified by the format string.

# C Syntax

ViStatus viPrintf(ViSession vi, ViString writeFmt, ...)

# Visual Basic Syntax

N/A
#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
writeFmt	IN	String describing the format for arguments.
	IN	Parameters to which the format string is applied.

## **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Para	meters were successfully formatted.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_IO	Could not perform write operation because of I/O error.
VI_ERROR_TMO	Timeout expired before write operation completed.
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>writeFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of

insufficient resources.

### Description

The viPrintf() operation sends data to a device as specified by the format string. Before sending the data, the operation formats the arguments in the parameter list as specified in the **writeFmt** string. The viWrite() operation performs the actual low-level I/O to the device. As a result, you should not use the viWrite() and viPrintf() operations in the same session.

The **writeFmt** string can include regular character sequences, special formatting characters, and special format specifiers. The regular characters (including white spaces) are written to the device unchanged. The special characters consist of '\' (backslash) followed by a character. The format specifier sequence consists of '%' (percent) followed by an optional modifier (flag), followed by a format code.

### **Special Formatting Characters**

Special formatting character sequences send special characters. The following table lists the special characters and describes what they send to the device.

Formatting Character	Character Sent to Device
\n	Sends the ASCII LF character. The END identifier will also be automatically sent.
\r	Sends an ASCII CR character.
\t	Sends an ASCII TAB character.
\###	Sends the ASCII character specified by the octal value.
\x##	Sends the ASCII character specified by the hexadecimal value.
\"	Sends the ASCII double-quote (") character.
\\	Sends a backslash (\) character.

### **Format Specifiers**

The format specifiers convert the next parameter in the sequence according to the modifier and format code, after which the formatted data is written to the specified device. The format specifier takes the following syntax:

%[modifiers]format code

where *format code* specifies which data type the argument is represented in. Modifiers are optional codes that describe the target data.

In the following tables, a 'd' format code refers to all conversion codes of type *integer* ('d', 'i', 'o', 'u', 'x', 'X'), unless specified as %d only. Similarly, an 'f' format code refers to all conversion codes of type *float* ('f', 'e', 'E', 'g', 'G'), unless specified as %f only.

Every conversion command starts with the % character and ends with a conversion character (format code). Between the % character and the format code, the following modifiers can appear in the sequence.

# **ANSI C Standard Modifiers**

Modifier	Supported with Format Code	Description
An integer specifying field width.	d, f, s format codes	This specifies the minimum field width of the converted argument. If an argument is shorter than the <i>field width</i> , it will be padded on the left (or the right if the - flag is present).
		Special case:
		For the @H, @Q, and @B flags, the <i>field width</i> includes the #H, #Q, and #B strings, respectively.
		An asterisk (*) may be present in lieu of a <i>field</i> <i>width</i> modifier, in which case an extra <b>arg</b> is used. This <b>arg</b> must be an integer representing the <i>field width</i> .
An integer specifying <i>precision</i> .	d, f, s format codes	The <i>precision</i> string consists of a string of decimal digits. A decimal point (.) must prefix the <i>precision</i> string. The <i>precision</i> string specifies the following:
		a. The minimum number of digits to appear for the @1, @H, @Q, and @B flags and the i, o, u,

x, and X format codes.

- b. The maximum number of digits after the decimal point in case of f format codes.
- c. The maximum numbers of characters for the string (s) specifier.
- d. Maximum significant digits for g format code.

An asterisk (\*) may be present in lieu of a *precision* modifier, in which case an extra **arg** is used. This **arg** must be an integer representing the *precision* of a numeric field.

An argument length modifier. h, l, L, z, and Z are legal values. (z and Z are not ANSI C standard modifiers.) h (d, b, B format codes) l (d, f, b, B format codes) L (f format code) z (b, B format codes) Z (b, B format codes) The argument length modifiers specify one of the following:

- a. The h modifier promotes the argument to a short or unsigned short, depending on the format code type.
- b. The 1 modifier promotes the argument to a long or unsigned long.
- c. The L modifier promotes the

argument to a long double parameter.

- d. The z modifier promotes the argument to an array of floats.
- e. The Z modifier promotes the argument to an array of doubles.

# **Enhanced Modifiers to ANSI C Standards**

Modifier	Supported with Format Code	Description
A comma (,) followed by an integer <i>n</i> , where <i>n</i> represents	%d (plus variants) and %f only	The corresponding argument is interpreted as a reference to the first element of an array of size <i>n</i> . The first <i>n</i> elements of this list are printed in the format specified by the format code.
the array size.		An asterisk (*) may be present after the <i>comma</i> (,) modifier, in which case an extra <b>arg</b> is used. This <b>arg</b> must be an integer representing the array size of the given type.
@1	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (for example, 123).
@2	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (for example, 123.45).
@3	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined NR3 compatible number. An NR3 number is

		a floating point number represented in an exponential form (for example, 1.2345E-67).
@H	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined <hexadecimal NUMERIC RESPONSE DATA&gt;. The number is represented in a base of sixteen form. Only capital letters should represent numbers. The number is of form #HXXX, where XXX is a hexadecimal number (for example, #HAF35B).</hexadecimal 
@Q	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined <octal numeric<br="">RESPONSE DATA&gt;. The number is represented in a base of eight form. The number is of the form #QYYY, where YYY is an octal number (for example, #Q71234).</octal>
@B	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined <binary numeric<br="">RESPONSE DATA&gt;. The number is represented in a base two form. The number is of the form #BZZZ, where ZZZ is a binary number (for example, #B011101001).</binary>

The following are the allowed format code characters. A format specifier sequence should include one and only one format code.

### **ANSI C Standard Format Codes**

% Send the ASCII percent (%) character. c Argument type: A character to be sent. d Argument type: An integer.

Modifier	Interpretation
Default functionality	Print an integer in NR1 format (an integer without a decimal point).
@2 or @3	The integer is converted into a floating point number and output in the correct format.
field width	Minimum field width of the output number. Any of the six IEEE 488.2 modifiers can also be specified with <i>field width</i> .
Length modifier	arg is a long integer.
Length modifier h	<b>arg</b> is a short integer.
,array size	<b>arg</b> points to an array of integers (or long or short integers, depending on the length modifier) of size array size. The elements of this array are separated by array size - 1 commas and output in the specified format.

f Argument type: A floating point number.

#### Modifier Interpretation

Default	Print a floating point number in NR2
Tunctionality	format (a number with at least one
	digit after the decimal point).

@1	Print an integer in NR1 format. The number is truncated.
@3	Print a floating point number in NR3 format (scientific notation). <i>Precision</i> can also be specified.
field width	Minimum field width of the output number. Any of the six IEEE 488.2 modifiers can also be specified with <i>field width</i> .
Length modifier l	<b>arg</b> is a double float.
Length modifier $L$	<b>arg</b> is a long double.
,array size	<b>arg</b> points to an array of floats (or doubles or long doubles, depending on the length modifier) of size array size. The elements of this array are separated by array size - 1 commas and output in the specified format.

s Argument type: A reference to a NULL-terminated string that is sent to the device without change.

### **Enhanced Format Codes**

b Argument type: A location of a block of data.

Flag or Modifier	Interpretation
Default functionality	The data block is sent as an IEEE 488.2 <definite length<br="">ARBITRARY BLOCK RESPONSE DATA&gt;. A count (long integer) must appear as a flag that specifies the number of elements (by default, bytes) in the block. A <i>field width</i> or <i>precision</i> modifier is not allowed with this format code.</definite>
* (asterisk)	An asterisk may be present instead of the count. In such a case, two <b>args</b> are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second <b>arg</b> is a reference to the data block. The size of an element is determined by the optional length modifier (see below), and the default is byte width.
Length modifier h	<b>arg</b> points to an array of unsigned short integers (16 bits). The count corresponds to the number of words rather than bytes. The data is swapped and padded into standard IEEE 488.2 format, if native computer representation is different.
Length	arg points to an array of unsigned long

Length<br/>modifier 1arg points to an array of unsigned long<br/>integers. The count specifies the<br/>number of longwords (32 bits). Each

longword data is swapped and padded into standard IEEE 488.2 format, if native computer representation is different.

- Length modifier z arg points to an array of floats. The count specifies the number of floating point numbers (32 bits). The numbers are represented in IEEE 754 format, if native computer representation is different.
- Length modifier Z arg points to an array of doubles. The count specifies the number of double floats (64 bits). The numbers will be represented in IEEE 754 format, if native computer representation is different.

B Argument type: A location of a block of data. The functionality is similar to b, except the data block is sent as an IEEE 488.2 <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>. This format involves sending an ASCII LF character with the END indicator set after the last byte of the block.

The END indicator is not appended when LF(n) is part of a binary data block, as with %b or %B.

y Argument type: A location of a block of binary data.

Modifier	Interpretation
Default functionality	The data block is sent as raw binary data. A count (long integer) must appear as a flag that specifies the number of elements (by default, bytes) in the block. A field width or precision modifier is not allowed with this format code.

\* (asterisk) An asterisk may be present instead

of the count. In such a case, two args are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second arg is a reference to the data block. The size of an element is determined by the optional length modifier (see below), and the default is byte width.

Length modifier h arg points to an array of unsigned short integers (16 bits). The count corresponds to the number of words rather than bytes. If the optional !ol byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate if native computer representation is different.

Length modifier l arg points to an array of unsigned long integers (32 bits). The count specifies the number of longwords rather than bytes. If the optional !ol byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate if native computer representation is different.

Length modifier <sup>z</sup> arg points to an array of floats. The count specifies the number of floating-point numbers (32 bits). If the optional !ol modifier is present, the data is sent in little endian format; otherwise, the data sent in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate to native computer format.

Length modifier	<b>arg</b> points to an array of doubles.
Z	The count specifies the number of
	double floats (64 bits). If the optional
	lol modifier is present, the data is
	sent in little endian format; otherwise,
	the data sent in standard IEEE 488.2
	format. The data will be byte
	swapped and padded as appropriate
	to native computer format.
Byte order	Data is sent in standard IEEE 488.2
modifier !ob	(hig endian) format. This is the

Syle order	Data is sent in standard IEEE 488.2
nodifier !ob	(big endian) format. This is the
	default behavior if neither !ob nor !ol
	is present.

Byte order	Data is sent in little endian format.
modifier !ol	

## **Other ANSI C Conversion Codes**

For ANSI C compatibility, VISA also supports the following conversion codes for output codes: 'i', 'o', 'u', 'n', 'x', 'X', 'e', 'E', 'g', 'G', and 'p'. For further explanation of these conversion codes, see the ANSI C Standard.

Also refer to your ANSI C documentation for information on the printf function.

**Note** VISA will not send out the data across the bus, by default, until a '\n' character is encountered in the format string (not the data stream). You can modify this behavior with the VI\_ATTR\_WR\_BUF\_OPER\_MODE attribute or with the viFlush() operation.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_WR\_BUF\_OPER\_MODE viFlush viScanf viSPrintf viVPrintf viVSPrintf

# viQueryf

### Purpose

Performs a formatted write and read through a single call to an operation.

# C Syntax

ViStatus viQueryf(ViSession vi, ViString writeFmt, ViString readFmt,...)

# Visual Basic Syntax

N/A

#### **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
writeFmt	IN	String describing the format of write arguments.
readFmt	IN	String describing the format of read arguments.
	IN/OUT	Parameters to which write and read format strings are applied.

### **Return Values**

Completion Codes Description				
VI_SUCCESS Su	ccessfully completed the query operation.			
Error Codes	Description			
VI_ERROR_INV_OBJECT	The given session reference is invalid.			
VI_ERROR_IO	Could not perform Read/Write operation because of I/O error.			
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.			
VI_ERROR_TMO	Timeout occurred before Read/Write operation completed.			
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> or <b>readFmt</b> string is invalid.			
VI_ERROR_NSUP_FMT	The format specifier is not supported for current argument type.			
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of			

insufficient resources.

# Description

This operation provides a mechanism of *Send*, *then receive* typical to a command sequence from a commander device. In this manner, the response generated from the command can be read immediately.

This operation is a combination of the viPrintf() and viScanf() operations. The first *n* arguments corresponding to the first format string are formatted by using the **writeFmt** string, then sent to the device. The write buffer is flushed immediately after the write portion of the operation completes. After these actions, the response data is read from the device into the remaining parameters (starting from parameter n + 1) using the **readFmt** string.



**Note** Because the prototype for this function cannot provide complete type-checking, remember that all output parameters must be passed by reference.

#### **Related Topics**

INSTR Resource SOCKET Resource viPrintf viScanf viVQueryf

# viRead

# Purpose

Reads data from device or interface synchronously.

# C Syntax

ViStatus viRead(ViSession **vi**, ViPBuf **buf**, ViUInt32 **count**, ViPUInt32 **retCount**)

# **Visual Basic Syntax**

viRead&(ByVal **vi**&, ByVal **buf**\$, ByVal **count**&, **retCount**&)

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	OUT	Location of a buffer to receive data from device.
count	IN	Number of bytes to be read.
retCount	OUT	Number of bytes actually transferred.

#### **Return Values**

#### **Completion Codes** Description The operation completed VI SUCCESS successfully and the END indicator was received (for interfaces that have END indicators). This completion code is returned regardless of whether the termination character is received or the number of bytes read is equal to **count**. The specified VI\_SUCCESS\_TERM\_CHAR

The specified termination character was read but no END indicator was received. This completion code is returned regardless of whether the number of bytes read is equal to **count**.

VI\_SUCCESS\_MAX\_CNT

The number of bytes read is equal to **count**. No END indicator was received and no termination character was read.

#### **Error Codes**

VI\_ERROR\_INV\_OBJECT

#### Description

The given session reference

is invalid.

VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.

#### VI\_ERROR\_INV\_SETUP

Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).

#### VI\_ERROR\_NCIC The interface associated with the given **vi** is not currently the controller in

#### VI\_ERROR\_NLISTENERS

No-listeners condition is detected (both NRFD and NDAC are unasserted).

transfer.

A framing error occurred during transfer.

charge.

VI_ERROR_ASRL_PARITY	A parity error
	occurred during

#### VI\_ERROR\_ASRL\_FRAMING

VI ERROR ASRL OVERRUN

An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.

#### VI\_ERROR\_IO

An unknown I/O error occurred during transfer.

#### VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.
## Description

The viRead() operation synchronously transfers data. The data read is to be stored in the buffer represented by **buf**. This operation returns only when the transfer terminates. Only one synchronous read operation can occur at any one time.

## **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viBufRead viReadAsync viReadToFile viWrite

# viReadAsync

## Purpose

Reads data from device or interface asynchronously.

## C Syntax

ViStatus viReadAsync(ViSession **vi**, ViPBuf **buf**, ViUInt32 **count**, ViPJobId **jobId**)

## Visual Basic Syntax

N/A

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

## **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	OUT	Location of a buffer to receive data from device.
count	IN	Number of bytes to be read.
jobld	OUT	Job ID of this asynchronous read operation.

## **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Asynchronous read operation successfully queued.
VI_SUCCESS_SYNC	Read operation performed synchronously.
Error Codes	Description
VI_ERROR_INV_OBJ	ECT The given session reference is invalid.
VI_ERROR_RSRC_LOO	CKED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_QUEUE_E	RROR Unable to queue read operation (usually due to the I/O completion event not being enabled or insufficient space in the session's queue).
VI_ERROR_IN_PROGI	RESS Unable to queue the asynchronous operation because there is already an operation in progress.

## Description

The viReadAsync() operation asynchronously transfers data. The data read is to be stored in the buffer represented by **buf**. This operation normally returns before the transfer terminates.

Before calling this operation, you should enable the session for receiving I/O completion events. After the transfer has completed, an I/O completion event is posted.

The operation returns **jobld**, which you can use with either viTerminate() to abort the operation, or with an I/O completion event to identify which asynchronous read operation completed. VISA will never return VI\_NULL for a valid **jobID**.



**Note** If you have enabled VI\_EVENT\_IO\_COMPLETION for queueing (VI\_QUEUE), for each successful call to viReadAsync(), you must call viWaitOnEvent() to retrieve the I/O completion event. This is true even if the I/O is done synchronously (that is, if the operation returns VI\_SUCCESS\_SYNC). If you are using LabVIEW, this is done for you automatically.

## **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_EVENT\_IO\_COMPLETION viEnableEvent viRead viTerminate viWaitOnEvent viWaitOnEvent

# viReadSTB

## Purpose

Reads a status byte of the service request.

## C Syntax

ViStatus viReadSTB(ViSession **vi**, ViPUInt16 **status**)

## Visual Basic Syntax

viReadSTB&(ByVal **vi**&, **status**%)

## **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR

#### **Parameters**

Name	Direction Description	
vi	IN	Unique logical identifier to a session.
status	OUT	Service request status byte.

## **Return Values**

values	
Completion Codes Descri	ption
VI_SUCCESS The	operation completed successfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKE	D Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_SRQ_NOCCURR	ED Service request has not been received for the session.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_	VIOL Violation of raw write protocol

occurred during transfer.

VI\_ERROR\_RAW\_RD\_PROT\_VIOL Violation of raw read protocol occurred during transfer.

> VI\_ERROR\_BERR Bus error occurred during transfer.

> > The interface associated with the given **vi** is not currently the controller in charge.

#### VI\_ERROR\_NLISTENERS

VI\_ERROR\_NCIC

No-listeners condition is detected (both NRFD and NDAC are unasserted).

#### VI\_ERROR\_INV\_SETUP

Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

#### VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.

## Description

## Status Bytes for 488.2 Instruments (GPIB, VXI, TCPIP, and USB)

This operation reads a service request status from a message-based device. The bus-specific details are:

- For a GPIB device, the status is read by serial polling the device.
- For a VXI device, VISA sends the Word Serial Read STB query.
- For a USB device, this function sends the READ\_STATUS\_BYTE command on the control pipe.

# Status Bytes for Non-488.2 Instruments (Serial INSTR, TCPIP SOCKET, and USB RAW)

A message is sent in response to a service request to retrieve status information. If VI\_ATTR\_IO\_PROT is VI\_PROT\_4882\_STRS, the device is sent the string "\*STB?\n", and then the device's status byte is read; otherwise, this operation is not valid.

Although the **status** output is a 16-bit value, the upper 8 bits are always 0. The lower 8 bits contain the actual status byte. For 488.2 instruments, this is the 488.2-defined status byte.

The IEEE 488.2 standard defines several bit assignments in the status byte. For example, if bit 6 of the **status** is set, the device is requesting service. In addition to setting bit 6 when requesting service, 488.2 devices also use two other bits to specify their status. Bit 4, the Message Available bit (MAV), is set when the device is ready to send previously queried data. Bit 5, the Event Status bit (ESB), is set if one or more of the enabled 488.2 events occurs. These events include power-on, user request, command error, execution error, device dependent error, query error, request control, and operation complete. The device can assert SRQ when ESB or MAV are set, or when a manufacturer-defined condition occurs. Manufacturers of 488.2 devices use the remaining lower-order bits to communicate the reason for the service request or to summarize the device state.

## **Related Topics**

INSTR Resource SOCKET Resource VI\_ATTR\_IO\_PROT VI\_EVENT\_SERVICE\_REQ

# viReadToFile

## Purpose

Read data synchronously, and store the transferred data in a file.

## C Syntax

ViStatus viReadToFile(ViSession **vi**, ViString **fileName**, ViUInt32 **count**, ViPUInt32 **retCount**)

## **Visual Basic Syntax**

viReadToFile& (ByVal **vi**&, ByVal **filename**\$, ByVal **count**&, **retCount**&)

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
fileName	IN	Name of file to which data will be written.
count	IN	Number of bytes to be read.
retCount	OUT	Number of bytes actually transferred.

#### **Return Values**

Completion Codes	Description
VI_SUCCESS	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count.

#### **Error Codes**

#### Description

The given session or object reference is invalid (both are the same value).

VI\_ERROR\_NSUP\_OPER The given **vi** does not support

VI\_ERROR\_RSRC\_LOCKED

VI ERROR INV OBJECT

Specified operation could not be performed because the resource identified by **vi** has been locked for this kind of

this operation.

access.

VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the

controller in charge.

VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error

VI_ERROR_ASKL_FARTT	A painty entit
	occurred during
	transfer.

VI\_ERROR\_ASRL\_FRAMING A framing error occurred during transfer.

#### VI\_ERROR\_ASRL\_OVERRUN

An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.

VI\_ERROR\_IO An unknown I/O error occurred

VI\_ERROR\_FILE\_ACCESS

An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.

during transfer.

#### VI\_ERROR\_FILE\_IO

An error occurred while accessing the specified file.

#### VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.

## Description

This read operation synchronously transfers data. The file specified in fileName is opened in binary write-only mode. If the value of VI\_ATTR\_FILE\_APPEND\_EN is VI\_FALSE, any existing contents are destroyed; otherwise, the file contents are preserved. The data read is written to the file. This operation returns only when the transfer terminates.

This operation is useful for storing raw data to be processed later.

#### Special Values for retCount Parameter

Value	Action Description	
VI_NULL	Do not return the number of bytes transferred.	

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_FILE\_APPEND\_EN viRead viWriteFromFile

## viScanf

## Purpose

Reads, converts, and formats data using the format specifier. Stores the formatted data in the parameters (designated by ...).

## C Syntax

ViStatus viScanf(ViSession vi, ViString readFmt, ...)

## Visual Basic Syntax

N/A

## **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

## Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
readFmt	IN	String describing the format for arguments.
	OUT	Parameters into which the data is read and the format string is applied.

## **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Dat	a was successfully read and formatted into parameter(s).
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_IO	Could not perform read operation because of I/O error.
VI_ERROR_TMO	Timeout expired before read operation completed.
VI_ERROR_INV_FMT	A format specifier in the <b>readFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>readFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of

insufficient resources.

## Description

The viScanf() operation receives data from a device, formats it by using the format string, and stores the resulting data in the **arg** parameter list. The viRead() operation is used for the actual low-level read from the device. As a result, you should not use the viRead() and viScanf() operations in the same session.



**Note** Because the prototype for this function cannot provide complete type-checking, remember that all output parameters must be passed by reference.

The format string can have format specifier sequences, white characters, and ordinary characters. The white characters—blank, vertical tabs, horizontal tabs, form feeds, new line/linefeed, and carriage return—are ignored except in the case of %c and %[]. All other ordinary characters except % should match the next character read from the device.

The format string consists of a %, followed by optional modifier flags, followed by one of the format codes in that sequence. It is of the form:

%[modifier]format code

where the optional modifier describes the data format, while format code indicates the nature of data (data type). One and only one format code should be performed at the specifier sequence. A format specification directs the conversion to the next input **arg**.

The results of the conversion are placed in the variable that the corresponding argument points to, unless the \* assignment-suppressing character is given. In such a case, no **arg** is used and the results are ignored.

The viScanf() operation accepts input until an END indicator is read or all the format specifiers in the **readFmt** string are satisfied. Thus, detecting an END indicator before the **readFmt** string is fully consumed will result in ignoring the rest of the format string. Also, if some data remains in the buffer after all format specifiers in the **readFmt** string are satisfied, the data will be kept in the buffer and will be used by the next viScanf() operation.

When viScanf() times out, the next call to viScanf() will read from an empty buffer and force a read from the device.

Notice that when an END indicator is received, not all arguments in the format string may be consumed. However, the operation still returns a successful completion code.

The following two tables describe optional modifiers that can be used in a format specifier sequence.
## **ANSI C Standard Modifiers**

Modifier	Supported with Format Code	Description
An integer representing the field width	%s, %c, % [] format codes	It specifies the maximum field width that the argument will take. A '#' may also appear instead of the integer <i>field width</i> , in which case the next <b>arg</b> is a reference to the <i>field width</i> . This <b>arg</b> is a reference to an integer for %c and %s. The <i>field width</i> is not allowed for %d or %f.
A length modifier ('h', 'l', 'L', 'z', or 'Z'). z and Z are not ANSI C standard modifiers.	h (d, b format codes) l (d, f, b format codes) L (f format code) z (b format code) Z (b format code)	The argument length modifiers specify one of the following: a. The h modifier promotes the argument to be a reference to a short integer or unsigned short integer, depending on the format code. b. The I modifier promotes the argument to point to a long integer or unsigned long integer.

c. The L modifier promotes the

argument to point to a long double floats parameter.

- d. The z modifier promotes the argument to point to an array of floats.
- e. The Z modifier promotes the argument to point to an array of double floats.

t An asterisk (\*) acts as the assignment suppression character. The input is not assigned to any parameters and is discarded.

\*

All format codes

## **Enhanced Modifiers to ANSI C Standards**

Modifier	Supported with Format Code	Description
A comma (,) followed by an integer <i>n</i> , where <i>n</i> represents	%d (plus variants) and %f only	The corresponding argument is interpreted as a reference to the first element of an array of size <i>n</i> . The first <i>n</i> elements of this list are printed in the format specified by the format code.
the array size.		A number sign (#) may be present after the comma (,) modifier, in which case an extra <b>arg</b> is used. This <b>arg</b> must be an integer representing the array size of the given type.
@1	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (for example, 123).
@2	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (for example, 123.45).
@H	%d (plus variants) and %f	Converts to an IEEE 488.2 defined <hexadecimal NUMERIC RESPONSE</hexadecimal 

	only	DATA>. The number is represented in a base of sixteen form. Only capital letters should represent numbers. The number is of form #HXXX, where XXX is a hexadecimal number (for example, #HAF35B).
@Q	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined <octal numeric<br="">RESPONSE DATA&gt;. The number is represented in a base of eight form. The number is of the form #QYYY, where YYY is an octal number (for example, #Q71234).</octal>
@B	%d (plus variants) and %f only	Converts to an IEEE 488.2 defined <binary numeric<br="">RESPONSE DATA&gt;. The number is represented in a base two form. The number is of the form #BZZZ, where ZZZ is a binary number (for example, #B011101001).</binary>

### **ANSI C Standard Format Codes**

c Argument type: A reference to a character.

Flags or Modifiers	Interpretation
Default functionality	A character is read from the device and stored in the parameter.
field width	<i>field width</i> number of characters are read and stored at the reference location (the default <i>field width</i> is 1). No NULL character is added at the end of the data block.



**Note** This format code does not ignore white space in the device input stream.

d Argument type: A reference to an integer.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until an entire number is read. The number read may be in either IEEE 488.2 formats <decimal numeric<br="">PROGRAM DATA&gt;, also known as NRf; flexible numeric representation (NR1, NR2, NR3); or <non- DECIMAL NUMERIC PROGRAM DATA&gt; (#H, #Q, and #B).</non- </decimal>
field width	The input number will be stored in a field at least this wide.
Length modifier 1	<b>arg</b> is a reference to a long integer.
Length	<b>arg</b> is a reference to a short integer.

- modifier h Rounding is performed according to IEEE 488.2 rules (0.5 and up).
- ,array size arg points to an array of integers (or long or short integers, depending on the length modifier) of size array size. The elements of this array should be separated by commas. Elements will be read until either array size number of elements are consumed or they are no longer separated by commas. If the array size contains a number sign (#), two arguments are used. The first arg read is a pointer to an integer specifying the maximum number of elements that the array can hold. The second **arg** should be a reference to an array. Also, the actual number of elements read is stored back in the first argument.

f Argument type: A reference to a floating point number.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until an entire number is read. The number read may be in either IEEE 488.2 formats <decimal numeric<br="">PROGRAM DATA&gt; (NRf) or <non- DECIMAL NUMERIC PROGRAM DATA&gt; (#H, #Q, and #B)</non- </decimal>
field width	The input will be stored in a field at least this wide.
Length modifier 1	<b>arg</b> is a reference to a double floating point number.

Length	arg is a reference to a long double
modifier L	number.

,array size arg points to an array of floats (or double or long double, depending on the length modifier) of size array size. The elements of this array should be separated by commas. Elements will be read until either array size number of elements are consumed or they are no longer separated by commas. If the array size contains a number sign (#), two arguments are used. The first arg read is a pointer to an integer specifying the maximum number of elements that the array can hold. The second arg should be a reference to an array. Also, the actual number of elements read is stored back in the first argument.

s Argument type: A reference to a string.

Flags or Modifiers	Interpretation
Default functionality	All leading white space characters are ignored. Characters are read from the device into the string until a white space character is read.
field width	This flag gives the maximum string size. If the <i>field width</i> contains a number sign (#), two arguments are used. The first argument read is a pointer to an integer specifying the maximum array size. The second should be a reference to an array. In case of <i>field width</i> characters already

read before encountering a white space, additional characters are read and discarded until a white space character is found. In case of *# field width*, the actual number of characters that were copied into the user array, not counting the trailing NULL character, are stored back in the integer pointed to by the first argument.

## **Enhanced Format Codes**

b Argument type: A reference to a data array.

Flags or Modifiers	Interpretation
Default functionality	The data must be in IEEE 488.2 <arbitrary block="" program<br="">DATA&gt; format. The format specifier sequence should have a flag describing the <i>field width</i>, which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the <i>field width</i> contains a # sign, two arguments are used. The first <b>arg</b> read is a pointer to a long integer specifying the maximum number of elements that the array can hold. The second <b>arg</b> should be a reference to an array. Also, the actual number of elements read is stored back in the first argument. In absence of length modifiers, the data is assumed to be of byte-size elements. In some cases, data might be read until an END indicator is read.</arbitrary>
Length modifier h	<b>arg</b> points to an array of 16-bit words, and count specifies the number of words. Data that is read is assumed to be in IEEE 488.2 byte ordering. It will be byte swapped and padded as appropriate to native computer format.

Length	<b>arg</b> points to an array of 32-bit
modifier 1	longwords, and count specifies the
	number of longwords. Data that is read

is assumed to be in IEEE 488.2 byte ordering. It will be byte swapped and padded as appropriate to native computer format.

Length	<b>arg</b> points to an array of floats, and
modifier z	count specifies the number of floating
	point numbers. Data that is read is an
	array of 32-bit IEEE 754 format floating
	point numbers.

Length modifier Z **arg** points to an array of doubles, and the count specifies the number of floating point numbers. Data that is read is an array of 64-bit IEEE 754 format floating point numbers.

t Argument type: A reference to a string.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until the first END indicator is received. The character on which the END indicator was received is included in the buffer.
field width	This flag gives the maximum string size. If an END indicator is not received before <i>field width</i> number of characters, additional characters are read and discarded until an END indicator arrives. <i>#field width</i> has the same meaning as in %s.

T Argument type: A reference to a string.

Flags or	Interpretation
Modifiers	interpretation

Default functionality	Characters are read from the device until the first linefeed character (\n) is received. The linefeed character is included in the buffer.

field width This flag gives the maximum string size. If a linefeed character is not received before *field width* number of characters, additional characters are read and discarded until a linefeed character arrives. *#field width* has the same meaning as in %s.

y Argument type: A location of a block of binary data.

Modifier	Interpretation
Default functionality	The data block is read as raw binary data. The format specifier sequence should have a flag describing the array size, which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the array size contains a # sign, two arguments are used. The first argument read is a pointer to a long integer that specifies the maximum number of elements that the array can hold. The second argument should be a reference to an array. Also, the actual number of elements read is stored back in the first argument. In absence of length modifiers, the data is assumed to be byte-size elements. In some cases, data might be read until an END indicator is read.

Length modifier h The data block is assumed to be a reference to an array of unsigned short integers (16 bits). The count corresponds to the number of words rather than bytes. If the optional !ol modifier is present, the data read is assumed to be in little endian format; otherwise, the data read is assumed to be in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate to native computer format.

Length modifier 1 The data block is assumed to be a reference to an array of unsigned long integers (32 bits). The count corresponds to the number of longwords rather than bytes. If the optional !ol modifier is present, the data read is assumed to be in little endian format; otherwise, the data read is assumed to be in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate to native computer format.

Length modifier z The data block is assumed to be a reference to an array of singleprecision floating-point numbers (32 bits). The count corresponds to the number of floats rather than bytes. If the optional !ol modifier is present, the data read is assumed to be in little endian format; otherwise, the data read is assumed to be in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate to native computer format.

- Length modifier The data block is assumed to be a reference to an array of double-Ζ precision floating-point numbers (64 bits). The count corresponds to the number of double floats rather than bytes. If the optional !ol modifier is present, the data read is assumed to be in little endian format; otherwise, the data read is assumed to be in standard IEEE 488.2 format. The data will be byte swapped and padded as appropriate to native computer format. The data being read is assumed to Byte order modifier !ob be in standard IEEE 488.2 (big endian) format. This is the default behavior if neither !ob nor !ol is present.
  - Byte orderThe data being read is assumed tomodifier !olbe in little endian format.

### **Other ANSI C Format Specifiers**

For ANSI C compatibility, VISA also supports the following format specifiers for input codes: 'i', 'o', 'u', 'n', 'x', 'X', 'e', 'E', 'g', 'G', 'p', '[...]', and '[^...]'. For further explanation of these conversion codes, see the ANSI C Standard.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_ATTR\_RD\_BUF\_OPER\_MODE viFlush viPrintf viSScanf viVScanf

# viSetAttribute

## Purpose

Sets the state of an attribute.

## C Syntax

ViStatus viSetAttribute(ViObject **vi**, ViAttr **attribute**, ViAttrState **attrState**)

## **Visual Basic Syntax**

viSetAttribute&(ByVal **vi**&, ByVal **attribute**&, ByVal **attrState**&)

### **Resource Classes**

All I/O session types

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
attribute	IN	Attribute for which the state is to be modified.
attrState	IN	The state of the attribute to be set for the specified object. The interpretation of the individual attribute value is defined by the object.

### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Attribute value set successfully.
VI_WARN_NSUP_ATTR_STATE	Although the specified attribute state is valid, it is not supported by this implementation.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced object.
VI_ERROR_NSUP_ATTR_STATE	The specified state of the attribute is not valid, or is not supported as defined by the object.
VI_ERROR_ATTR_READONLY	The specified attribute is Read Only.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource

identified by vi has been locked for this kind of access.

### Description

The viSetAttribute() operation is used to modify the state of an attribute for the specified object.

Both VI\_WARN\_NSUP\_ATTR\_STATE and

VI\_ERROR\_NSUP\_ATTR\_STATE indicate that the specified attribute state is not supported. A resource normally returns the error code VI\_ERROR\_NSUP\_ATTR\_STATE when it cannot set a specified attribute state. The completion code VI\_WARN\_NSUP\_ATTR\_STATE is intended to alert the application that although the specified optional attribute state is not supported, the application should not fail. One example is attempting to set an attribute value that would increase performance speeds. This is different than attempting to set an attribute value that specifies required but nonexistent hardware (such as specifying a VXI ECL trigger line when no hardware support exists) or a value that would change assumptions a resource might make about the way data is stored or formatted (such as byte order).

Some attributes documented as being generally Read/Write may at times be Read Only. This is usually the case when an attribute configures how the VISA driver receives events of a given type, and the event type associated with that attribute is currently enabled. Under these circumstances, calling viSetAttribute on that attribute returns VI\_ERROR\_ATTR\_READONLY.

The error code VI\_ERROR\_RSRC\_LOCKED is returned only if the specified attribute is Read/Write and Global, and the resource is locked by another session.

#### **Related Topics**

Attributes viGetAttribute VISA Resource Template

# viSetBuf

### Purpose

Sets the size for the formatted I/O and/or low-level I/O communication buffer(s).

## C Syntax

ViStatus viSetBuf(ViSession vi, ViUInt16 mask, ViUInt32 size)

## **Visual Basic Syntax**

viSetBuf&(ByVal **vi**&, ByVal **mask**%, ByVal **size**&)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mask	IN	Specifies the type of buffer.
size	IN	The size to be set for the specified buffer(s).

### **Return Values**

Completion Codes	Description
VI_SUCCESS	Buffer size set successfully.
VI_WARN_NSUP_BUF	The specified buffer is not supported.
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given session reference is invalid.
VI_ERROR_RSRC_LOC	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_ALLOC	The system could not allocate the buffer(s) of the specified <b>size</b> because of insufficient resources.
VI_ERROR_INV_MAS	5K The system cannot set the buffer for the given <b>mask</b> .

### Description

The viSetBuf() operation changes the buffer size of the read and/or write buffer for formatted I/O and/or serial communication. The **mask** parameter specifies the buffer for which to set the size. The **mask** parameter can specify multiple buffers by bit-ORing any of the following values together.

Flags	Interpretation
VI_READ_BUF (1)	Formatted I/O read buffer.
VI_WRITE_BUF (2)	Formatted I/O write buffer.
VI_IO_IN_BUF (16)	Low-level I/O receive buffer.
VI_IO_OUT_BUF (32)	Low-level I/O transmit buffer.

A call to viSetBuf() flushes the session's related Read/Write buffer(s). 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.

Since not all serial drivers support user-defined buffer sizes, it is possible that a specific implementation of VISA may not be able to control this feature. If an application requires a specific buffer size for performance reasons, but a specific implementation of VISA cannot guarantee that size, then it is recommended to use some form of handshaking to prevent overflow conditions.

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 Topics**

Automatically Flushing the Formatted I/O Buffers Controlling the Serial 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 INSTR Resource INTFC Resource Manually Flushing the Formatted I/O Buffers Recommendations for Using the VISA Buffers SERVANT Resource SOCKET Resource VI\_ATTR\_RD\_BUF\_SIZE VI\_ATTR\_WR\_BUF\_SIZE viFlush viPrintf

# viSPrintf

### Purpose

Converts, formats, and sends the parameters (designated by...) to a user-specified buffer as specified by the format string.

## C Syntax

ViStatus viSPrintf(ViSession vi, ViPBuf buf, ViString writeFmt, ...)

## Visual Basic Syntax

N/A

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	OUT	Buffer where data is to be written.
writeFmt	IN	The format string to apply to parameters in ViVAList.
	IN	Parameters to which the format string is applied. The formatted data is written to the specified buf.

### **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Para	meters were successfully formatted.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>writeFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.
### Description

The viSPrintf() operation is similar to viPrintf(), except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

If this operation outputs an END indicator before all the arguments are satisfied, then the rest of the **writeFmt** string is ignored and the buffer string is still terminated by a NULL.



**Note** The size of the **buf** parameter should be large enough to hold the formatted I/O contents plus the NULL termination character.

#### **Related Topics**

**INSTR Resource** 

**INTFC Resource** 

SERVANT Resource

SOCKET Resource

<u>viPrintf</u>

<u>viSScanf</u>

<u>viVPrintf</u>

<u>viVSPrintf</u>

# viSScanf

#### Purpose

Reads, converts, and formats data from a user-specified buffer using the format specifier. Stores the formatted data in the parameters (designated by ...).

## C Syntax

ViStatus viSScanf(ViSession **vi**, ViBuf **buf**, ViString **readFmt**, **..**)

# Visual Basic Syntax

N/A

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR, VXI SERVANT

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Buffer from which data is read and formatted.
readFmt	IN	String describing the format for arguments.
	OUT	Parameters into which the data is read and the format string is applied.

### **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Dat	a was successfully read and formatted into parameter(s).
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the <b>readFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>readFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

### Description

The viSScanf() operation is similar to viScanf(), except that the data is read from a user-specified buffer rather than from a device.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource

SOCKET Resource

<u>viScanf</u>

<u>viSPrintf</u>

<u>viVScanf</u>

<u>viVSScanf</u>

# viStatusDesc

### Purpose

Returns a user-readable description of the status code passed to the operation.

## C Syntax

ViStatus viStatusDesc(ViObject vi, ViStatus status, ViChar desc[])

## Visual Basic Syntax

viStatusDesc&(ByVal **vi**&, ByVal **status**&, ByVal **desc**\$)

### **Resource Classes**

All I/O session types, all event object types, VISA Resource Manager

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
status	IN	Status code to interpret.
desc	OUT	The user-readable string interpretation of the status code passed to the operation.

#### **Return Values**

**Completion Codes** 

#### Description

VI\_SUCCESS

Description successfully returned.

VI\_WARN\_UNKNOWN\_STATUS

The status code passed to the operation could not be interpreted.

## Description

The viStatusDesc() operation is used to retrieve a user-readable string that describes the status code presented. If the string cannot be interpreted, the operation returns the warning code VI\_WARN\_UNKNOWN\_STATUS. However, the output string **desc** is valid regardless of the status return value.



**Note** The size of the **desc** parameter should be *at least* 256 bytes.

#### **Related Topics**

**Completion Codes** 

Error Codes

VISA Resource Template

# viTerminate

### Purpose

Requests a VISA session to terminate normal execution of an operation.

## C Syntax

ViStatus viTerminate(ViObject **vi**, ViUInt16 **degree**, ViJobId **jobId**)

# Visual Basic Syntax

N/A

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, GPIB-VXI MEMACC, GPIB-VXI BACKPLANE, PXI INSTR, PXI MEMACC, PXI BACKPLANE, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI MEMACC, VXI BACKPLANE, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
degree	IN	VI_NULL (0).
jobld	IN	Specifies an operation identifier.

### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Request serviced successfully.
Error Codes	Description
VI_ERROR_INV_OBJ	ECT The given object reference is invalid.
VI_ERROR_INV_JOB	_ID Specified job identifier is invalid.
VI_ERROR_INV_DEG	REE Specified <b>degree</b> is invalid.

## Description

This operation is used to request a session to terminate normal execution of an operation, as specified by the **jobld** parameter. The **jobld** parameter is a unique value generated from each call to an asynchronous operation.

If a user passes VI\_NULL as the **jobld** value to viTerminate(), VISA will abort any calls in the current process executing on the specified **vi**. Any call that is terminated this way should return VI\_ERROR\_ABORT. Due to the nature of multi-threaded systems, for example where operations in other threads may complete normally before the operation viTerminate() has any effect, the specified return value is not guaranteed.

#### **Related Topics**

VI\_EVENT\_IO\_COMPLETION viMoveAsync/viMoveAsyncEx viReadAsync VISA Resource Template viWriteAsync

# viUninstallHandler

## Purpose

Uninstalls handlers for events.

## C Syntax

ViStatus viUninstallHandler(ViSession **vi**, ViEventType **eventType**, ViHndlr **handler**, ViAddr **userHandle**)

# Visual Basic Syntax

N/A

### **Resource Classes**

All I/O session types

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
eventType	IN	Logical event identifier.
handler	IN	Interpreted as a valid reference to a handler to be uninstalled by a client application.
userHandle	IN	A value specified by an application that can be used for identifying handlers uniquely in a session for an event.

#### **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Event handler successfully uninstalled.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT The given session

reference is invalid.

VI_ERROR_INV_EVENT	Specified event
	type is not
	supported by the
	resource.

VI_ERROR_INV_HNDLR_RE	EF Either the specified handler reference or the user context value (or both) does not match any installed handler.
	ED A bandlar is not

VI\_ERROR\_HNDLR\_NINSTALLED A handler is not currently installed for the specified event.

## Description

The viUninstallHandler() operation allows applications to uninstall handlers for events on sessions. Applications should also specify the value in the **userHandle** parameter that was passed while installing the handler. VISA identifies handlers uniquely using the handler reference and this value. All the handlers, for which the handler reference and the value matches, are uninstalled. Specifying VI\_ANY\_HNDLR as the value for the **handler** parameter causes the operation to uninstall all the handlers with the matching value in the **userHandle** parameter.

**Note** Calling viUninstallHandler() removes the specified handler from the list of active handlers on the given session. If no handlers remain for the specified event type, the VISA driver disables that event type on the given session. It is not valid for a user to call this operation from within a callback, because this may cause a deadlock condition within the VISA driver.

#### **Related Topics**

N

viDisableEvent viEventHandler viInstallHandler VISA Resource Template

# viUnlock

## Purpose

Relinquishes a lock for the specified resource.

# C Syntax

ViStatus viUnlock(ViSession vi)

# Visual Basic Syntax

viUnlock&(ByVal **vi**&)

### **Resource Classes**

All I/O session types

#### Parameters

Name	Direction Description	
vi	IN	Unique logical identifier to a session.

**Return Values** 

Completion CodesDescriptionVI\_SUCCESSLock<br/>successfully<br/>relinquished.VI\_SUCCESS\_NESTED\_EXCLUSIVECall succeeded,<br/>but this session<br/>still has nested<br/>exclusive locks.VI\_SUCCESS\_NESTED\_SHAREDCall succeeded,<br/>but this session<br/>still has nested<br/>exclusive locks.

#### **Error Codes**

#### Description

VI\_ERROR\_INV\_OBJECT

The given session reference is invalid.

VI\_ERROR\_SESN\_NLOCKED The current session did not have any lock on the resource.

### Description

This operation is used to relinquish the lock previously obtained using the viLock() operation.

### **Related Topics**

<u>viLock</u>

VISA Resource Template
# viUnmapAddress

# Purpose

Unmaps memory space previously mapped by viMapAddress().

# C Syntax

ViStatus viUnmapAddress(ViSession **vi**)

# Visual Basic Syntax

viUnmapAddress&(ByVal **vi**&)

#### **Resource Classes**

FireWire INSTR, GPIB-VXI INSTR, GPIB-VXI MEMACC, PXI INSTR, PXI MEMACC, VXI INSTR, VXI MEMACC

### Parameters

Name	Direction Description	
vi	IN	Unique logical identifier to a session.

## **Return Values**

Completion Codes	Description
VI_SUCCESS	Operation completed successfully.
Error Codes	Description
VI_ERROR_INV_O	BJECT The given session reference is invalid.
VI_ERROR_NSUP_	OPER The given <b>vi</b> does not support this operation.
VI_ERROR_WINDOW_	NMAPPED The specified session is not currently mapped.

# Description

The viUnmapAddress() operation unmaps the region previously mapped by the viMapAddress() or viMapAddressEx() operation for this session.

### **Related Topics**

INSTR Resource MEMACC Resource viMapAddress/viMapAddressEx

# viUnmapTrigger

## Purpose

Undo a previous map from the specified trigger source line to the specified destination line.

# C Syntax

ViStatus viUnmapTrigger(ViSession **vi**, ViInt16 **trigSrc**, ViInt16 **trigDest**)

# **Visual Basic Syntax**

viUnmapTrigger& (ByVal **vi**&, ByVal **trigSrc**%, ByVal **trigDest**%)

# **Resource Classes**

GPIB-VXI BACKPLANE, PXI BACKPLANE, VXI BACKPLANE

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
trigSrc	IN	Source line used in previous map. Refer to the <i>Description</i> section for actual values.
trigDest	IN	Destination line used in previous map. Refer to the <i>Description</i> section for actual values.

# **Return Values**

Completion Codes D	escription
VI_SUCCESS	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given session reference is invalid.
VI_ERROR_NSUP_OPE	ER The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCH	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_LIN	E One of the specified lines ( <b>trigSrc</b> or <b>trigDest</b> ) is invalid.
VI_ERROR_TRIG_NMAP	PED The path from <b>trigSrc</b> to <b>trigDest</b> is not currently mapped.
VI_ERROR_NSUP_LIN	IE One of the specified lines ( <b>trigSrc</b> or <b>trigDest</b> ) is not supported by this VISA implementation.

## Description

This operation can be used to undo a previous mapping of one trigger line to another. This operation is valid only on BACKPLANE (mainframe) sessions.

#### Special Values for trigSrc Parameter

Value	Action Description
VI_TRIG_TTL0 - VI_TRIG_TTL7	Unmap the specified VXI TTL trigger line.
VI_TRIG_ECL0 - VI_TRIG_ECL1	Unmap the specified VXI ECL trigger line.
VI_TRIG_PANEL_IN	Unmap the controller's front panel trigger input line.
VI_TRIG_PANEL_OUT	Unmap the controller's front panel trigger output line.

#### **Special Values for trigDest Parameter**

Value	Action Description
VI_TRIG_TTL0 - VI_TRIG_TTL7	Unmap the specified VXI TTL trigger line.
VI_TRIG_ECL0 - VI_TRIG_ECL1	Unmap the specified VXI ECL trigger line.
VI_TRIG_PANEL_IN	Unmap the controller's front panel trigger input line.
VI_TRIG_PANEL_OUT	Unmap the controller's front panel trigger output line.
VI_TRIG_ALL	Unmap all trigger lines to which <b>trigSrc</b> is currently connected.

This operation unmaps only one trigger mapping per call. In other words, if viMapTrigger() was called multiple times on the same BACKPLANE Resource and created multiple mappings for either **trigSrc** or **trigDest**, trigger mappings other than the one specified by **trigSrc** and **trigDest** should remain in effect after this call completes.

#### **Related Topics**

**BACKPLANE Resource** 

viMapTrigger

# viUsbControlIn

### Purpose

Performs a USB control pipe transfer from the device.



**Note** This operation is intended only for users familiar with the USB protocol.

# C Syntax

ViStatus viUsbControlIn (ViSession **vi**, ViInt16 **bmRequestType**, ViInt16 **bRequest**, ViUInt16 **wValue**, ViUInt16 **wIndex**, ViUInt16 **wLength**, ViPBuf **buf**, ViPUInt16 **retCnt)**;

## **Visual Basic Syntax**

viUsbControlIn& (ByVal **vi**&, ByVal **bmRequestType**%, ByVal **bRequest**%, ByVal **wValue**%, ByVal **wIndex**%, ByVal **wLength**%, buf **As Byte**, **retCnt**%)

## **Resource Classes**

USB INSTR, USB RAW

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
bmRequestType	IN	The <b>bmRequestType</b> parameter of the setup stage of a USB control transfer. Refer to the USB specification for further details.
bRequest	IN	The <b>bRequest</b> parameter of the setup stage of a USB control transfer.
wValue	IN	The <b>wValue</b> parameter of the setup stage of a USB control transfer.
wIndex	IN	The <b>windex</b> parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
wLength	IN	The <b>wLength</b> parameter of the setup stage of a USB control transfer. This value also specifies the size of the data buffer to receive the data from the optional data stage of the control transfer.
buf	OUT	The data buffer that receives the data from the optional data stage of the control

transfer. This is ignored if **wLength** is 0.

retCntOUTNumber of bytes actually<br/>transferred in the optional<br/>data stage of the control<br/>transfer. This parameter may<br/>be VI\_NULL if you do not<br/>need this information.

Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.
VI_ERROR_INV_PARAMETER	The value of some parameter—which parameter is not known—is invalid.
VI_ERROR_INV_MASK	The bmRequestType parameter contains an invalid mask.

# Description

The viUsbControlIn() operation synchronously performs a USB control pipe transfer from the device. The values of the data payload in the setup stage of the control transfer are taken as parameters and include **bmRequestType**, **bRequest**, **wValue**, **wIndex**, and **wLength**. An optional data buffer **buf** receives data if a data stage is required for this transfer. Only one USB control pipe transfer operation can occur at any one time.

#### **Related Topics**

INSTR Resource RAW Resource VI\_ATTR\_USB\_CTRL\_PIPE viUsbControlOut

# viUsbControlOut

## Purpose

Performs a USB control pipe transfer to the device.



**Note** This operation is intended only for users familiar with the USB protocol.

# C Syntax

ViStatus viUsbControlOut (ViSession **vi**, ViInt16 **bmRequestType**, ViInt16 **bRequest**, ViUInt16 **wValue**, ViUInt16 **wIndex**, ViUInt16 **wLength**, ViPBuf **buf**);

## **Visual Basic Syntax**

viUsbControlOut& (ByVal **vi**&, ByVal **bmRequestType**%, ByVal **bRequest**%, ByVal **wValue**%, ByVal **wIndex**%, ByVal **wLength**%, buf **As Byte**)

## **Resource Classes**

USB INSTR, USB RAW

#### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
bmRequestType	IN	The <b>bmRequestType</b> parameter of the setup stage of a USB control transfer. Refer to the USB specification for further details.
bRequest	IN	The <b>bRequest</b> parameter of the setup stage of a USB control transfer.
wValue	IN	The <b>wValue</b> parameter of the setup stage of a USB control transfer.
wIndex	IN	The <b>windex</b> parameter of the setup stage of a USB control transfer. This is usually the index of the interface or endpoint.
wLength	IN	The <b>wLength</b> parameter of the setup stage of a USB control transfer. This value also specifies the size of the data buffer that contains the data to send in the optional data stage of the control transfer.
buf	IN	The data buffer that sends the data in the optional data

stage of the control transfer. This is ignored if **wLength** is 0.

Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.
VI_ERROR_INV_PARAMETER	The value of some parameter—which parameter is not known—is invalid.
VI_ERROR_INV_MASK	The bmRequestType parameter contains an invalid mask.

# Description

The viUsbControlOut() operation synchronously performs a USB control pipe transfer to the device. The values of the data payload in the setup stage of the control transfer are taken as parameters and include **bmRequestType**, **bRequest**, **wValue**, **windex**, and **wLength**. An optional data buffer **buf** contains the data to send if a data stage is required for this transfer. Only one USB control pipe transfer operation can occur at any one time.

#### **Related Topics**

INSTR Resource RAW Resource VI\_ATTR\_USB\_CTRL\_PIPE viUsbControlln

# viVPrintf

### Purpose

Converts, formats, and sends the parameters designated by **params** to the device or interface as specified by the format string.

# C Syntax

ViStatus viVPrintf(ViSession vi, ViString writeFmt, ViVAList params)

# **Visual Basic Syntax**

viVPrintf&(ByVal **vi**&, ByVal **writeFmt**\$, **params** as Any)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
writeFmt	IN	String describing the format to apply to <b>params</b> .
params	IN	A list containing the variable number of parameters on which the format string is applied. The formatted data is written to the specified device.
# **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Para	meters were successfully formatted.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_IO	Could not perform write operation because of I/O error.
VI_ERROR_TMO	Timeout expired before write operation completed.
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>writeFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of

insufficient resources.

## Description

This operation is similar to viPrintf(), except that the **params** parameters list provides the parameters rather than separate **arg** parameters.

### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viPrintf viSPrintf viVScanf viVSPrintf

# viVQueryf

# Purpose

Performs a formatted write and read through a single call to an operation.

# C Syntax

ViStatus viVQueryf(ViSession **vi**, ViString **writeFmt**, ViString **readFmt**, ViVAList **params**)

## **Visual Basic Syntax**

viVQueryf&(ByVal **vi**&, ByVal **writeFmt**\$, ByVal **readFmt**\$, **params** as Any)

### **Resource Classes**

GPIB INSTR, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
writeFmt	IN	String describing the format of write arguments.
readFmt	IN	String describing the format of read arguments.
params	IN/OUT	A list containing the variable number of write and read parameters. The write parameters are formatted and written to the specified device. The read parameters store the data read from the device after the format string is applied to the data.

# **Return Values**

Completion Codes Des	cription
VI_SUCCESS Su	ccessfully completed the query operation.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKEI	Specified operation could not be performed because the resource identified by vi has been locked for this kind of access.
VI_ERROR_IO	Could not perform Read/Write operation because of I/O error.
VI_ERROR_TMO	Timeout occurred before Read/Write operation completed.
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> or <b>readFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	The format specifier is not supported for current argument type.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of

insufficient resources.

# Description

This operation is similar to viQueryf(), except that the **params** parameters list provides the parameters rather than the separate **arg** parameter list



**Note** Because the prototype for this function cannot provide complete type-checking, remember that all output parameters must be passed by reference.

### **Related Topics**

INSTR Resource SOCKET Resource viQueryf

# viVScanf

### Purpose

Reads, converts, and formats data using the format specifier. Stores the formatted data in the parameters designated by **params**.

# C Syntax

ViStatus viVScanf(ViSession vi, ViString readFmt, ViVAList params)

# **Visual Basic Syntax**

viVScanf&(ByVal **vi**&, ByVal **readFmt**\$, **params** as Any)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
readFmt	IN	String describing the format to apply to <b>params</b> .
params	OUT	A list with the variable number of parameters into which the data is read and the format string is applied.

# **Return Values**

Completion Codes De	scription
VI_SUCCESS E	Data was successfully read and formatted into <b>params</b> parameter(s).
Error Codes	Description
VI_ERROR_INV_OBJEC	Γ The given session reference is invalid.
VI_ERROR_RSRC_LOCK	ED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_IO	Could not perform read operation because of I/O error.
VI_ERROR_TMO	Timeout expired before read operation completed.
VI_ERROR_INV_FMT	A format specifier in the <b>readFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>readFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of

insufficient resources.

# Description

This operation is similar to viScanf(), except that the **params** parameters list provides the parameters rather than separate **arg** parameters.



**Note** Because the prototype for this function cannot provide complete type-checking, remember that all output parameters must be passed by reference.

### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viScanf viSScanf viVPrintf viVSScanf

# viVSPrintf

### Purpose

Converts, formats, and sends the parameters designated by **params** to a user-specified buffer as specified by the format string.

# C Syntax

ViStatus viVSPrintf(ViSession **vi**, ViPBuf **buf**, ViString **writeFmt**, ViVAList **params**)

## **Visual Basic Syntax**

viVSPrintf&(ByVal **vi**&, ByVal **buf**\$, ByVal **writeFmt**\$, **params** as Any)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	OUT	Buffer where data is to be written.
writeFmt	IN	The format string to apply to parameters in ViVAList.
params	IN	A list containing the variable number of parameters on which the format string is applied. The formatted data is written to the specified <b>buf</b> .

# **Return Values**

Completion Codes Desc	ription
VI_SUCCESS Para	meters were successfully formatted.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the <b>writeFmt</b> string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the <b>writeFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

# Description

This operation is similar to viVPrintf(), except that the output is not written to the device; it is written to the user-specified buffer. This output buffer is NULL terminated.

If this operation outputs an END indicator before all the arguments are satisfied, then the rest of the **writeFmt** string is ignored and the buffer string is still terminated by a NULL.



**Note** The size of the **buf** parameter should be large enough to hold the formatted I/O contents plus the NULL termination character.

### **Related Topics**

**INSTR Resource** 

**INTFC Resource** 

SERVANT Resource

SOCKET Resource

<u>viPrintf</u>

<u>viSPrintf</u>

<u>viVPrintf</u>

<u>viVSScanf</u>

# viVSScanf

### Purpose

Reads, converts, and formats data from a user-specified buffer using the format specifier. Stores the formatted data in the parameters designated by **params**.

# C Syntax

ViStatus viVSScanf(ViSession **vi**, ViBuf **buf**, ViString **readFmt**, ViVAList **params**)

# **Visual Basic Syntax**

viVSScanf&(ByVal **vi**&, ByVal **buf**\$, ByVal **readFmt**\$, **params** as Any)

### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Buffer from which data is read and formatted.
readFmt	IN	String describing the format to apply to <b>params</b> .
params	OUT	A list with the variable number of parameters into which the data is read and the format string is applied.

# **Return Values**

Completion Codes D	escription
VI_SUCCESS	Data was successfully read and formatted into <b>params</b> parameter(s).
Error Codes	Description
VI_ERROR_INV_OBJE	CT The given session reference is invalid.
VI_ERROR_RSRC_LOCF	KED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_INV_FM7	A format specifier in the <b>readFmt</b> string is invalid.
VI_ERROR_NSUP_FM	T A format specifier in the <b>readFmt</b> string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

# Description

The viVSScanf() operation is similar to viVScanf(), except that the data is read from a user-specified buffer rather than a device.



**Note** Because the prototype for this function cannot provide complete type checking, remember that all output parameters must be passed by reference.

### **Related Topics**

INSTR Resource

INTFC Resource SERVANT Resource SOCKET Resource viScanf viSScanf viVScanf viVSprintf

# viVxiCommandQuery

### Purpose

Sends the device a miscellaneous command or query and/or retrieves the response to a previous query.

# C Syntax

ViStatus viVxiCommandQuery(ViSession **vi**, ViUInt16 **mode**, ViUInt32 **cmd**, ViPUInt32 **response**)

# **Visual Basic Syntax**

viVxiCommandQuery&(ByVal vi&, ByVal mode%, ByVal cmd&, response&)
## **Resource Classes**

GPIB-VXI INSTR, VXI INSTR

#### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
mode	IN	Specifies whether to issue a command and/or retrieve a response. Refer to the Description section for actual values.
cmd	IN	The miscellaneous command to send.
response	OUT	The response retrieved from the device. If the mode specifies to send a command rather than retrieve a response, you can use VI_NULL for this parameter.

## **Return Values**

Vuldes	
Completion Codes Description	I
VI_SUCCESS The oper suc	ation completed ccessfully.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw

read protocol

occurred during transfer.

VI\_ERROR\_OUTP\_PROT\_VIOL Device reported an output protocol error during transfer.

VI\_ERROR\_INP\_PROT\_VIOL Device reported an input protocol error during transfer.

#### VI\_ERROR\_BERR Bus error occurred during transfer.

#### VI\_ERROR\_RESP\_PENDING

A previous response is still pending, causing a multiple query error.

VI_ERROR_INV_MODE	The value
	specified by the
	mode parameter
	is invalid.

## Description

The viVxiCommandQuery() operation can send a command or query, or receive a response to a query previously sent to the device. The **mode** parameter specifies whether to issue a command and/or retrieve a response, and indicates the type or size of command and/or response to use. The following table defines the values for the **mode** parameter.

Mode	Action Description
VI_VXI_CMD16	Send 16-bit Word Serial command.
VI_VXI_CMD16_RESP16	Send 16-bit Word Serial query; get 16-bit response.
VI_VXI_RESP16	Get 16-bit response from previous query.
VI_VXI_CMD32	Send 32-bit Word Serial command.
VI_VXI_CMD32_RESP16	Send 32-bit Word Serial query; get 16-bit response.
VI_VXI_CMD32_RESP32	Send 32-bit Word Serial query; get 32-bit response.
VI_VXI_RESP32	Get 32-bit response from previous query.

Notice that the **mode** you specify can cause all or part of the **cmd** or **response** parameters to be ignored.

- If **mode** specifies sending a 16-bit command, the upper half of **cmd** is ignored.
- If mode specifies retrieving a response only, cmd is ignored.
- If **mode** specifies sending a command only, **response** is ignored. You can use VI\_NULL for the value of **response**.

• If **mode** specifies to retrieve a 16-bit value, the upper half of **response** is set to 0.

Refer to the VXI Specification for defined Word Serial commands.

#### **Related Topics**

**INSTR Resource** 

# viWaitOnEvent

### Purpose

Waits for an occurrence of the specified event for a given session.

## C Syntax

ViStatus viWaitOnEvent(ViSession **vi**, ViEventType **inEventType**, ViUInt32 **timeout**, ViPEventType **outEventType**, ViPEvent **outContext**)

## **Visual Basic Syntax**

viWaitOnEvent&(ByVal **vi**&, ByVal **inEventType**&, ByVal **timeout**&, **outEventType**&, **outContext**&)

## **Resource Classes**

All I/O session types

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
inEventType	IN	Logical identifier of the event(s) to wait for.
timeout	IN	Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds.
outEventType	OUT	Logical identifier of the event actually received.
outContext	OUT	A handle specifying the unique occurrence of an event.

### **Return Values**

Completion Codes	Description
VI_SUCCESS	Wait terminated successfully on receipt of an event occurrence. The queue is empty.
VI_SUCCESS_QUEUE_NEMPTY	Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence of the type specified by <b>inEventType</b> available for this session.
VI_WARN_QUEUE_OVERFLOW	The event returned is valid. One or more events that occurred have not been raised because there was no room available on the queue at the time of their occurrence. This could happen because VI_ATTR_MAX_QUEUE_LENGTH is not set to a large enough value for your application and/or events are coming in faster than you are servicing them.
Error Codes	Description
VI_ERROR_INV_OBJI	ECT The given session reference is invalid.

VI\_ERROR\_INV\_EVENT

Specified event type is not supported by the resource.

VI_ERROR_TMO	Specified event did not occur within the specified time period.
VI_ERROR_NENABLED	The session must be enabled for events of the specified type in order to receive them.
VI_ERROR_QUEUE_OVERFLOW	No new event is raised because there is no room

#### raised because there is no room available on the queue. This means you have already received all previous events but not closed them. You must call viClose on each event you receive from viWaitOnEvent.

# Description

The viWaitOnEvent() operation suspends the execution of a thread of an application and waits for an event of the type specified by **inEventType** for a time period specified by **timeout**. You can wait only for events that have been enabled with the viEnableEvent() operation. Refer to individual event descriptions for context definitions. If the specified **inEventType** is VI\_ALL\_ENABLED\_EVENTS, the operation waits for any event that is enabled for the given session. If the specified timeout value is VI\_TMO\_INFINITE, the operation is suspended indefinitely. If the specified timeout value is VI\_TMO\_INFINITE, the operation be used to dequeue events from an event queue.

When the **outContext** handle returned from a successful invocation of viWaitOnEvent() is no longer needed, it should be passed to viClose().

If a session's event queue becomes full and a new event arrives, the new event is discarded. The default event queue size (per session) is 50, which is sufficiently large for most applications. If an application expects more than 50 events to arrive without having been handled, it can modify the value of the attribute VI\_ATTR\_MAX\_QUEUE\_LENGTH to the required size.

The **outEventType** and **outContext** parameters are optional and can be VI\_NULL. This can be used if the event type is known from the **inEventType** parameter, or if the **outContext** handle is not needed to retrieve additional information. If VI\_NULL is used for the **outContext** parameter, VISA will automatically close the event context.

### **Related Topics**

Events VI\_ATTR\_MAX\_QUEUE\_LENGTH viClose viEnableEvent VISA Resource Template

# viWrite

# Purpose

Writes data to device or interface synchronously.

# C Syntax

ViStatus viWrite(ViSession **vi**, ViBuf **buf**, ViUInt32 **count**, ViPUInt32 **retCount**)

# **Visual Basic Syntax**

viWrite&(ByVal **vi**&, ByVal **buf**\$, ByVal **count**&, **retCount**&)

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Location of a data block to be sent to a device.
count	IN	Number of bytes to be written.
retCount	OUT	Number of bytes actually transferred.

## **Return Values**

Completion Codes Description	
VI_SUCCESS Transfe	r completed.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during

transfer.

VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.
VI_ERROR_NLISTENERS	No-listeners condition is detected (both NRFD and NDAC are unasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

VI\_ERROR\_CONN\_LOST The I/O

connection for the given session has been lost.

## Description

The viWrite() operation synchronously transfers data. The data to be written is in the buffer represented by **buf**. This operation returns only when the transfer terminates. Only one synchronous write operation can occur at any one time.

### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viBufWrite viRead viWriteAsync viWriteFromFile

# viWriteAsync

# Purpose

Writes data to device or interface asynchronously.

# C Syntax

ViStatus viWriteAsync(ViSession **vi**, ViBuf **buf**, ViUInt32 **count**, ViPJobId **jobId**)

# Visual Basic Syntax

N/A

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### **Parameters**

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
buf	IN	Location of a data block to be sent to a device.
count	IN	Number of bytes to be written.
jobld	OUT	Job ID of this asynchronous write operation.

## **Return Values**

<b>Completion Codes</b>	Description
VI_SUCCESS	Asynchronous write operation successfully queued.
VI_SUCCESS_SYNC	Write operation performed synchronously.
Error Codes	Description
VI_ERROR_INV_OBJ	ECT The given session reference is invalid.
VI_ERROR_RSRC_LOO	CKED Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_QUEUE_E	RROR Unable to queue write operation (usually due to the I/O completion event not being enabled or insufficient space in the session's queue).
VI_ERROR_IN_PROGI	RESS Unable to queue the asynchronous operation because there is already an operation in progress.

# Description

The viWriteAsync() operation asynchronously transfers data. The data to be written is in the buffer represented by **buf**. This operation normally returns before the transfer terminates.

Before calling this operation, you should enable the session for receiving I/O completion events. After the transfer has completed, an I/O completion event is posted.

The operation returns a job identifier that you can use with either viTerminate() to abort the operation or with an I/O completion event to identify which asynchronous write operation completed. VISA will never return VI\_NULL for a valid **jobId**.



**Note** If you have enabled VI\_EVENT\_IO\_COMPLETION for queueing (VI\_QUEUE), for each successful call to viWriteAsync(), you must call viWaitOnEvent() to retrieve the I/O completion event. This is true even if the I/O is done synchronously (that is, if the operation returns VI\_SUCCESS\_SYNC). If you are using LabVIEW, this is done for you automatically.

### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource VI\_EVENT\_IO\_COMPLETION viEnableEvent viReadAsync viTerminate viWaitOnEvent viWaitOnEvent

# viWriteFromFile

# Purpose

Take data from a file and write it out synchronously.

# C Syntax

ViStatus viWriteFromFile(ViSession **vi**, ViString **fileName**, ViUInt32 **count**, ViPUInt32 **retCount**)

# **Visual Basic Syntax**

viWriteFromFile& (ByVal vi&, ByVal filename\$, ByVal count&, retCount&)

#### **Resource Classes**

GPIB INSTR, GPIB INTFC, GPIB-VXI INSTR, Serial INSTR, TCPIP INSTR, TCPIP SOCKET, USB INSTR, USB RAW, VXI INSTR, VXI SERVANT

### Parameters

Name	Direction	Description
vi	IN	Unique logical identifier to a session.
fileName	IN	Name of file from which data will be read.
count	IN	Number of bytes to be written.
retCount	OUT	Number of bytes actually transferred.

## **Return Values**

Completion Codes Description	
VI_SUCCESS Transfe	r completed.
Error Codes	Description
VI_ERROR_INV_OBJECT	The given session reference is invalid.
VI_ERROR_NSUP_OPER	The given <b>vi</b> does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <b>vi</b> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during
transfer.

VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given <b>vi</b> is not currently the controller in charge.
VI_ERROR_NLISTENERS	No-listeners condition is detected (both NRFD and NDAC are unasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_FILE_ACCESS	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
VI_ERROR_FILE_IO	An error occurred

while accessing the specified file.

## VI\_ERROR\_CONN\_LOST

The I/O connection for the given session has been lost.

## Description

This write operation synchronously transfers data. The file specified in **fileName** is opened in binary read-only mode, and the data (up to end-of-file or the number of bytes specified in count) is read. The data is then written to the device. This operation returns only when the transfer terminates.

This operation is useful for sending data that was already processed and/or formatted.

#### Special Values for retCount Parameter

Value	Action Description
VI_NULL	Do not return the number of
	bytes transferred.

If you pass VI\_NULL as the **retCount** parameter to the viWriteFromFile() operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

#### **Related Topics**

INSTR Resource INTFC Resource SERVANT Resource SOCKET Resource viReadToFile viWrite

# Resources

This section lists the attributes, events, and operations in each resource in VISA.

Refer to <u>Attributes</u>, <u>Events</u>, or <u>Operations</u> for more detailed information.

# **VISA Resource Template**

This section lists the attributes, events, and operations for the VISA Resource Template. The attributes, events, and operations in the VISA Resource Template are available to all other resources.

### Attributes

VI ATTR MAX QUEUE LENGTH VI ATTR RM SESSION VI ATTR RSRC\_CLASS VI ATTR RSRC\_IMPL\_VERSION VI ATTR RSRC\_LOCK STATE VI ATTR RSRC\_MANF\_ID VI ATTR RSRC\_MANF\_NAME VI ATTR RSRC\_NAME VI ATTR RSRC\_NAME VI ATTR RSRC\_SPEC\_VERSION VI ATTR USER DATA/VI ATTR USER DATA 32/VI ATTR USER DATA

#### **Events**

VI\_EVENT\_EXCEPTION

# Operations

viClose (vi) viDisableEvent (vi, eventType, mechanism) viDiscardEvents (vi, eventType, mechanism) viEnableEvent (vi, eventType, mechanism, context) viGetAttribute (vi, attribute, attrState) viInstallHandler (vi, eventType, handler, userHandle) viLock (vi, lockType, timeout, requestedKey, accessKey) viSetAttribute (vi, attribute, attrState) viStatusDesc (vi, status, desc) viTerminate (vi, degree, jobId) viUninstallHandler (vi, eventType, handler, userHandle) viUninstallHandler (vi, eventType, handler, userHandle) viUninstallHandler (vi, inEventType, timeout, outEventType, outContext)

# **VISA Resource Manager**

This section lists the attributes, events, and operations for the VISA Resource Manager. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the operations listed below.

## Attributes

The attributes for the VISA Resource Template are available to this resource. This resource has no defined attributes of its own.

## **Events**

None

# Operations

viFindNext (findList, instrDesc) viFindRsrc (sesn, expr, findList, retcnt, instrDesc) viOpen (sesn, rsrcName, accessMode, timeout, vi) viOpenDefaultRM (sesn) viParseRsrc(sesn, rsrcName, intfType, intfNum) viParseRsrcEx (sesn, rsrcName, intfType, intfNum, rsrcClass, unaliasdExpandedRsrcName, aliasIfExists)

# **INSTR Resource**

This section lists the attributes, events, and operations for the INSTR Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

### Attributes

VI ATTR ASRL ALLOW TRANSMIT VI ATTR ASRL AVAIL NUM VI ATTR ASRL BAUD VI ATTR ASRL BREAK LEN VI ATTR ASRL BREAK STATE VI ATTR ASRL CTS STATE VI ATTR ASRL DATA BITS VI ATTR ASRL DCD STATE VI ATTR ASRL DISCARD NULL VI ATTR ASRL DSR STATE VI ATTR ASRL DTR STATE VI ATTR ASRL END IN VI ATTR ASRL END OUT VI ATTR ASRL FLOW CONTROL VI ATTR ASRL PARITY VI ATTR ASRL REPLACE CHAR VI ATTR ASRL RI STATE VI ATTR ASRL RTS STATE VI ATTR ASRL STOP BITS VI ATTR ASRL WIRE MODE VI ATTR ASRL XOFF CHAR VI ATTR ASRL XON CHAR VI ATTR CMDR LA VI ATTR DEST ACCESS PRIV VI ATTR DEST BYTE ORDER VI ATTR DEST INCREMENT VI ATTR DMA ALLOW EN

VI ATTR FDC CHNL VI ATTR FDC MODE VI ATTR FDC USE PAIR VI ATTR FILE APPEND EN VI ATTR GPIB PRIMARY ADDR VI ATTR GPIB READDR EN VI ATTR GPIB REN STATE VI ATTR GPIB SECONDARY ADDR VI ATTR GPIB UNADDR EN VI ATTR IMMEDIATE SERV VI ATTR INTF INST NAME VI ATTR INTF NUM **VI ATTR INTF PARENT NUM** VI ATTR INTF TYPE VI ATTR IO PROT VI ATTR MAINFRAME LA VI ATTR MANF ID VI ATTR MANF NAME VI ATTR MEM BASE/VI ATTR MEM BASE 32/VI ATTR MEM BASE VI ATTR MEM SIZE/VI ATTR MEM SIZE 32/VI ATTR MEM SIZE 64 VI ATTR MEM SPACE VI ATTR MODEL CODE VI ATTR MODEL NAME VI ATTR RD BUF OPER MODE VI ATTR SEND END EN VI ATTR SLOT VI ATTR SRC ACCESS PRIV VI ATTR SRC BYTE ORDER VI ATTR SRC INCREMENT

VI ATTR SUPPRESS END EN VI ATTR TCPIP ADDR VI ATTR TCPIP DEVICE NAME VI\_ATTR\_TCPIP\_HOSTNAME VI ATTR TERMCHAR VI ATTR TERMCHAR EN VI ATTR TMO VALUE VI ATTR TRIG ID VI ATTR VXI DEV CLASS VI ATTR VXI LA VI ATTR VXI TRIG SUPPORT VI ATTR WIN ACCESS VI ATTR WIN ACCESS PRIV VI ATTR WIN BASE ADDR/VI ATTR WIN BASE ADDR 32/VI ATTR VI ATTR WIN BYTE ORDER VI ATTR WIN SIZE/VI ATTR WIN SIZE 32/VI ATTR WIN SIZE 64 VI ATTR WR BUF OPER MODE

### **Events**

VI\_EVENT\_IO\_COMPLETION VI\_EVENT\_SERVICE\_REQ VI\_EVENT\_TRIG VI\_EVENT\_VXI\_SIGP VI\_EVENT\_VXI\_VME\_INTR

## Operations

<u>viAssertTrigger</u> (vi, protocol) viBufRead (vi, buf, count, retCount) <u>viBufWrite</u> (vi, buf, count, retCount) viClear (vi) viFlush (vi, mask) viGpibControlREN (vi, mode) viIn8 (vi, space, offset, val8) viIn8Ex (vi, space, offset, val8) viIn16 (vi, space, offset, val16) viIn16Ex (vi, space, offset, val16) viIn32 (vi, space, offset, val32) viIn32Ex (vi, space, offset, val32) viIn64 (vi, space, offset, val64) viIn64Ex (vi, space, offset, val64) viMapAddress (vi, mapSpace, mapBase, mapSize, access, suggested, address) viMapAddressEx (vi, mapSpace, mapBase, mapSize, access, suggested, address) viMemAlloc (vi, size, offset) <u>viMemAllocEx</u> (vi, size, offset) viMemFree (vi, offset) viMemFreeEx (vi, offset) viMove (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth,

length)

viMoveEx (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

viMoveAsync (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveAsyncEx (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveIn8 (vi, space, offset, length, buf8) viMoveIn8Ex (vi, space, offset, length, buf8) viMoveIn16 (vi, space, offset, length, buf16) viMoveIn16Ex (vi, space, offset, length, buf16) viMoveIn32 (vi, space, offset, length, buf32) viMoveIn32Ex (vi, space, offset, length, buf32) viMoveIn64 (vi, space, offset, length, buf64) viMoveIn64Ex (vi, space, offset, length, buf64) viMoveOut8 (vi, space, offset, length, buf8) viMoveOut8Ex (vi, space, offset, length, buf8) viMoveOut16 (vi, space, offset, length, buf16) viMoveOut16Ex (vi, space, offset, length, buf16) viMoveOut32 (vi, space, offset, length, buf32) viMoveOut32Ex (vi, space, offset, length, buf32) viMoveOut64 (vi, space, offset, length, buf64) viMoveOut64Ex (vi, space, offset, length, buf64) viOut8 (vi, space, offset, val8) viOut8Ex (vi, space, offset, val8) viOut16 (vi, space, offset, val16) viOut16Ex (vi, space, offset, val16) viOut32 (vi, space, offset, val32) viOut32Ex (vi, space, offset, val32) viOut64 (vi, space, offset, val64) viOut64Ex (vi, space, offset, val64) viPeek8 (vi, addr, val8) viPeek16 (vi, addr, val16) viPeek32 (vi, addr, val32) viPeek64 (vi, addr, val64) viPoke8 (vi, addr, val8)

viPoke16 (vi, addr, val16) viPoke32 (vi, addr, val32) viPoke64 (vi, addr, val64) viPrintf (vi, writeFmt, ...) viQuervf (vi, writeFmt, readFmt, ...) viRead (vi, buf, count, retCount) viReadAsync (vi, buf, count, jobId) viReadSTB (vi, status) viReadToFile (vi, fileName, count, retCount) viScanf (vi, readFmt, ...) viSetBuf (vi, mask, size) viSPrintf (vi, buf, writeFmt, ...) viSScanf (vi, buf, readFmt, ...) viUnmapAddress (vi) viUsbControlIn viUsbControlOut viVPrintf (vi, writeFmt, params) viVQueryf (vi, writeFmt, readFmt, params) viVScanf (vi, readFmt, params) viVSPrintf (vi, buf, writeFmt, params) viVSScanf (vi, buf, readFmt, params) viVxiCommandQuery (vi, mode, cmd, response) viWrite (vi, buf, count, retCount) viWriteAsync(vi, buf, count, jobId)

# **MEMACC Resource**

This section lists the attributes, events, and operations for the MEMACC Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

## Attributes

VI ATTR DEST ACCESS PRIV VI ATTR DEST BYTE ORDER VI ATTR DEST INCREMENT VI ATTR DMA ALLOW EN VI ATTR GPIB PRIMARY ADDR VI ATTR GPIB SECONDARY ADDR VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF PARENT NUM VI ATTR INTF TYPE VI ATTR SRC ACCESS PRIV VI ATTR SRC BYTE ORDER VI ATTR SRC INCREMENT VI ATTR TMO VALUE VI ATTR VXI LA VI ATTR WIN ACCESS VI ATTR WIN ACCESS PRIV VI ATTR WIN BASE ADDR/VI ATTR WIN BASE ADDR 32/VI ATTR VI ATTR WIN BYTE ORDER VI ATTR WIN SIZE/VI ATTR WIN SIZE 32/VI ATTR WIN SIZE 64

### **Events**

VI\_EVENT\_IO\_COMPLETION

## Operations

viln8 (vi, space, offset, val8)

viln8Ex (vi, space, offset, val8)

viIn16 (vi, space, offset, val16)

viIn16Ex (vi, space, offset, val16)

viIn32 (vi, space, offset, val32)

viIn32Ex (vi, space, offset, val32)

viIn64 (vi, space, offset, val64)

viIn64Ex (vi, space, offset, val64)

viMapAddress (vi, mapSpace, mapBase, mapSize, access, suggested, address)

viMapAddressEx (vi, mapSpace, mapBase, mapSize, access, suggested, address)

viMemAlloc (vi, size, offset)

viMemAllocEx (vi, size, offset)

<u>viMove</u> (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

viMoveEx (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

viMoveAsync (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveAsyncEx (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveIn8 (vi, space, offset, length, buf8)

viMoveIn8Ex (vi, space, offset, length, buf8)

viMoveIn16 (vi, space, offset, length, buf16)

viMoveIn16Ex (vi, space, offset, length, buf16)

viMoveIn32 (vi, space, offset, length, buf32)

viMoveIn32Ex (vi, space, offset, length, buf32)

viMoveIn64 (vi, space, offset, length, buf64)

viMoveIn64Ex (vi, space, offset, length, buf64)

viMoveOut8 (vi, space, offset, length, buf8) viMoveOut8Ex (vi, space, offset, length, buf8) viMoveOut16 (vi, space, offset, length, buf16) viMoveOut16Ex (vi, space, offset, length, buf16) viMoveOut32 (vi, space, offset, length, buf32) viMoveOut32Ex (vi, space, offset, length, buf32) viMoveOut64 (vi, space, offset, length, buf64) viMoveOut64Ex (vi, space, offset, length, buf64) viOut8 (vi, space, offset, val8) viOut8Ex (vi, space, offset, val8) viOut16 (vi, space, offset, val16) viOut16Ex (vi, space, offset, val16) viOut32 (vi, space, offset, val32) viOut32Ex (vi, space, offset, val32) viOut64 (vi, space, offset, val64) viOut64Ex (vi, space, offset, val64) viPeek8 (vi, addr, val8) viPeek16 (vi, addr, val16) viPeek32 (vi, addr, val32) viPeek64 (vi, addr, val64) viPoke8 (vi, addr, val8) viPoke16 (vi, addr, val16) viPoke32 (vi, addr, val32) viPoke64 (vi, addr, val64) viUnmapAddress (vi)

# **INTFC Resource**

This section lists the attributes, events, and operations for the INTFC Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

### **Attributes**

VI ATTR DEV STATUS BYTE VI ATTR EVENT TYPE VI ATTR FILE APPEND EN VI ATTR GPIB ATN STATE VI ATTR GPIB CIC STATE VI ATTR GPIB HS488 CBL LEN VI ATTR GPIB NDAC STATE VI ATTR GPIB PRIMARY ADDR VI ATTR GPIB REN STATE VI ATTR GPIB SECONDARY ADDR VI ATTR GPIB SRQ STATE VI ATTR GPIB SYS CNTRL STATE VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF TYPE VI ATTR MAX QUEUE LENGTH VI ATTR RD BUF OPER MODE VI\_ATTR\_RM\_SESSION VI ATTR RSRC IMPL VERSION VI ATTR RSRC LOCK STATE VI ATTR RSRC MANF ID VI ATTR RSRC MANF NAME VI ATTR RSRC NAME VI ATTR RSRC SPEC VERSION VI ATTR SEND END EN VI ATTR TERMCHAR VI ATTR TERMCHAR EN

VI\_ATTR\_TMO\_VALUE VI\_ATTR\_TRIG\_ID VI\_ATTR\_USER\_DATA/VI\_ATTR\_USER\_DATA\_32/VI\_ATTR\_USER\_DATA VI\_ATTR\_WR\_BUF\_OPER\_MODE

#### **Events**

VI\_EVENT\_CLEAR VI\_EVENT\_GPIB\_CIC VI\_EVENT\_GPIB\_LISTEN VI\_EVENT\_GPIB\_TALK VI\_EVENT\_IO\_COMPLETION VI\_EVENT\_SERVICE\_REQ VI\_EVENT\_TRIG

## Operations

**<u>ViAssertTrigger</u>** (vi, protocol) viBufRead (vi, buf, count, retCount) viBufWrite (vi, buf, count, retCount) viFlush (vi, mask) viGpibCommand (vi, buf, count, retCount) viGpibControlATN (vi, mode) viGpibControlREN (vi, mode) viGpibPassControl (vi, primAddr, secAddr) viGpibSendIFC (vi) viPrintf (vi, writeFmt, ...) viRead (vi, buf, count, retCount) viReadAsync (vi, buf, count, jobId) viReadToFile (vi, fileName, count, retCount) viScanf (vi, readFmt, ...) viSetBuf (vi, mask, size) viSPrintf (vi, buf, writeFmt, ...) viSScanf (vi, buf, readFmt, ...) viVPrintf (vi, writeFmt, params) viVScanf (vi, readFmt, params) viVSPrintf (vi, buf, writeFmt, params) viVSScanf (vi, buf, readFmt, params) viWrite (vi, buf, count, retCount) viWriteAsync (vi, buf, count, jobId) viWriteFromFile (vi, fileName, count, retCount)

# **BACKPLANE** Resource

This section lists the attributes, events, and operations for the BACKPLANE Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

### Attributes

VI ATTR GPIB PRIMARY ADDR VI ATTR GPIB SECONDARY ADDR VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF PARENT NUM VI ATTR INTF TYPE VI ATTR MAINFRAME LA VI\_ATTR\_PXI\_CHASSIS VI\_ATTR\_PXI\_DEST\_TRIG\_BUS VI ATTR PXI SRC TRIG BUS VI ATTR PXI TRIG BUS VI ATTR TMO VALUE VI ATTR TRIG ID VI ATTR VXI TRIG STATUS VI ATTR VXI TRIG SUPPORT VI ATTR VXI VME INTR STATUS VI ATTR VXI VME SYSFAIL STATE

### **Events**

VI\_EVENT\_TRIG VI\_EVENT\_VXI\_VME\_SYSFAIL VI\_EVENT\_VXI\_VME\_SYSRESET

## Operations

viAssertIntrSignal (vi, mode, statusID) viAssertTrigger (vi, protocol) viAssertUtilSignal (vi, line) viMapTrigger (vi, trigSrc, trigDest, mode) viUnmapTrigger (vi, trigSrc, trigDest)

# **SERVANT Resource**

This section lists the attributes, events, and operations for the SERVANT Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

### Attributes

VI ATTR CMDR LA VI ATTR DEV STATUS BYTE VI ATTR DMA ALLOW EN VI ATTR FILE APPEND EN VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF TYPE VI ATTR IO PROT VI ATTR RD BUF OPER MODE VI ATTR SEND END EN VI ATTR TERMCHAR VI\_ATTR\_TERMCHAR\_EN VI\_ATTR\_TMO\_VALUE VI ATTR TRIG ID VI ATTR VXI LA VI ATTR VXI VME SYSFAIL STATE VI ATTR WR BUF OPER MODE
#### **Events**

VI\_EVENT\_CLEAR VI\_EVENT\_IO\_COMPLETION VI\_EVENT\_TRIG VI\_EVENT\_VXI\_VME\_SYSRESET

### Operations

viAssertIntrSignal (vi, mode, statusID) viAssertUtilSignal (vi, line) viBufRead (vi, buf, count, retCount) viBufWrite (vi, buf, count, retCount) viFlush (vi, mask) viPrintf (vi, writeFmt, ...) viRead (vi, buf, count, retCount) viReadAsync (vi, buf, count, jobId) viReadToFile (vi, fileName, count, retCount) viScanf (vi, readFmt, ...) viSetBuf (vi, mask, size) viSPrintf (vi, buf, writeFmt, ...) viSScanf (vi, buf, readFmt, ...) viVPrintf (vi, writeFmt, params) viVScanf (vi, readFmt, params) viVSPrintf (vi, buf, writeFmt, params) viVSScanf (vi, buf, readFmt, params) viWrite (vi, buf, count, retCount) viWriteAsync (vi, buf, count, jobId) viWriteFromFile (vi, fileName, count, retCount)

### **SOCKET Resource**

This section lists the attributes, events, and operations for the SOCKET Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

#### Attributes

VI ATTR FILE APPEND EN VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF TYPE VI ATTR IO PROT VI ATTR RD BUF OPER MODE VI ATTR SEND END EN VI\_ATTR\_TCPIP\_ADDR VI\_ATTR\_TCPIP\_HOSTNAME VI ATTR TCPIP KEEPALIVE VI ATTR TCPIP NODELAY VI\_ATTR\_TCPIP\_PORT VI\_ATTR\_TERMCHAR VI\_ATTR\_TERMCHAR\_EN VI ATTR TMO VALUE VI\_ATTR\_TRIG\_ID VI ATTR WR BUF OPER MODE

#### **Events**

VI\_EVENT\_IO\_COMPLETION

### Operations

viAssertTrigger (vi, protocol) viBufRead (vi, buf, count, retCount) viBufWrite (vi, buf, count, retCount) viClear (vi) viFlush (vi, mask) viPrintf (vi, writeFmt, ...) viQueryf (vi, writeFmt, readFmt, ...) viRead (vi, buf, count, retCount) viReadAsync (vi, buf, count, jobId) viReadSTB (vi, status) viReadToFile (vi, fileName, count, retCount) viScanf (vi, readFmt, ...) viSetBuf (vi, mask, size) viSPrintf (vi, buf, writeFmt, ...) viSScanf (vi, buf, readFmt, ...) viVPrintf (vi, writeFmt, params) viVQueryf (vi, writeFmt, readFmt, params) viVScanf (vi, readFmt, params) viVSPrintf (vi, buf, writeFmt, params) viVSScanf (vi, buf, readFmt, params) viWrite (vi, buf, count, retCount) viWriteAsync (vi, buf, count, jobId) viWriteFromFile (vi, fileName, count, retCount)

## **RAW Resource**

This section lists the attributes, events, and operations for the RAW Resource. The attributes, events, and operations in the VISA Resource Template are available to this resource in addition to the attributes and operations listed below.

#### Attributes

VI ATTR FILE APPEND EN VI ATTR INTF INST NAME VI ATTR INTF NUM VI ATTR INTF TYPE VI ATTR IO PROT VI ATTR MANF ID VI ATTR MANF NAME VI ATTR MAX QUEUE LENGTH VI ATTR MODEL CODE VI ATTR MODEL NAME VI ATTR RD BUF OPER MODE VI ATTR RD BUF SIZE VI ATTR RSRC CLASS VI ATTR RSRC IMPL VERSION VI ATTR RSRC LOCK STATE VI ATTR RSRC MANF ID VI ATTR RSRC MANF NAME VI\_ATTR\_RSRC\_NAME VI ATTR RSRC SPEC VERSION VI ATTR SUPPRESS END EN VI ATTR TERMCHAR VI ATTR TERMCHAR EN VI ATTR TMO VALUE VI ATTR USB ALT SETTING VI ATTR USB BULK IN PIPE VI ATTR USB BULK IN STATUS VI ATTR USB BULK OUT PIPE

VI\_ATTR\_USB\_BULK\_OUT\_STATUS VI\_ATTR\_USB\_CLASS VI\_ATTR\_USB\_END\_IN VI\_ATTR\_USB\_END\_IN VI\_ATTR\_USB\_INTFC\_NUM VI\_ATTR\_USB\_INTR\_IN\_PIPE VI\_ATTR\_USB\_INTR\_IN\_STATUS VI\_ATTR\_USB\_MAX\_INTR\_SIZE VI\_ATTR\_USB\_NUM\_INTFCS VI\_ATTR\_USB\_NUM\_PIPES VI\_ATTR\_USB\_PROTOCOL VI\_ATTR\_USB\_SERIAL\_NUM VI\_ATTR\_USB\_SERIAL\_NUM VI\_ATTR\_USB\_SUBCLASS VI\_ATTR\_USER\_DATA/VI\_ATTR\_USER\_DATA\_32/VI\_ATTR\_USER\_DATA VI\_ATTR\_WR\_BUF\_OPER\_MODE

#### **Events**

VI\_EVENT\_EXCEPTION VI\_EVENT\_IO\_COMPLETION VI\_EVENT\_USB\_INTR

### Operations

<u>viAssertTrigger</u> <u>viBufRead</u> <u>viBufWrite</u> viClear <u>viClose</u> <u>viDisableEvent</u> <u>viDiscardEvents</u> <u>viEnableEvent</u> <u>viEventHandler</u> <u>viFindNext</u> <u>viFindRsrc</u> <u>viGetAttribute</u> <u>viInstallHandler</u> <u>viLock</u> <u>viOpenDefaultRM</u> <u>viOpen</u> <u>viParseRsrc</u> <u>viParseRsrcEx</u> <u>viPrintf</u> <u>viReadAsync</u> <u>viReadSTB</u> viReadToFile viRead <u>viScanf</u> <u>viSetAttribute</u> <u>viTerminate</u> <u>viUninstallHandler</u> viUnlock
viUsbControlInd
viUsbControlOut
viVPrintf
viVScanf
viVSPrintf
viVSScanf
viUsbControlInd
viUsbControlInd
viUsbControlOut
viWaitOnEvent
viWriteAsync
viWriteFromFile
viWrite

# **Completion Codes**

This topic lists and describes the completion codes.

Completion Codes	Values	Meaning
VI_SUCCESS	0	Operation completed s
VI_SUCCESS_EVENT_EN	3FFF0002h	Specified event is alre for at least one of the s mechanisms.
VI_SUCCESS_EVENT_DIS	3FFF0003h	Specified event is alre disabled for at least or specified mechanisms
VI_SUCCESS_QUEUE_EMPTY	3FFF0004h	Operation completed s but queue was already
VI_SUCCESS_TERM_CHAR	3FFF0005h	The specified terminat character was read.
VI_SUCCESS_MAX_CNT	3FFF0006h	The number of bytes r to the input count.
VI_WARN_QUEUE_OVERFLOW	3FFF000Ch	The event returned is y or more events that oc not been raised becau was no room available queue at the time of th occurrence. This could because VI_ATTR_MAX_QUEL is not set to a large en for your application an are coming in faster th servicing them.
VI_WARN_CONFIG_NLOADED	3FFF0077h	The specified configur does not exist or coulc loaded; using VISA-sp defaults.
VI_SUCCESS_DEV_NPRESENT	3FFF007Dh	Session opened succe the device at the speci

address is not respond

		•
VI_SUCCESS_TRIG_MAPPED	3FFF007Eh	The path from trigSrc t already mapped.
VI_SUCCESS_QUEUE_NEMPTY	3FFF0080h	Wait terminated succe receipt of an event not There is still at least or event occurrence of th type(s) available for th
VI_WARN_NULL_OBJECT	3FFF0082h	The specified object re uninitialized.
VI_WARN_NSUP_ATTR_STATE	3FFF0084h	Although the specified attribute is valid, it is n supported by this reso implementation.
VI_WARN_UNKNOWN_STATUS	3FFF0085h	The status code passe operation could not be
VI_WARN_NSUP_BUF	3FFF0088h	The specified buffer is supported.
VI_SUCCESS_NCHAIN	3FFF0098h	Event handled succes not invoke any other h this session for this ev
VI_SUCCESS_NESTED_SHARED	3FFF0099h	Operation completed s and this session has n shared locks.
VI_SUCCESS_NESTED_EXCLUSIVE	3FFF009Ah	Operation completed s and this session has n exclusive locks.
VI_SUCCESS_SYNC	3FFF009Bh	Asynchronous operation was actually performed synchronously.
VI_WARN_EXT_FUNC_NIMPL	3FFF00A9h	The operation succeed lower level driver did n implement the extende

functionality.

### **Error Codes**

This topic lists and describes the error codes.

Completion Codes	Values	Meaning
VI_ERROR_SYSTEM_ERROR	BFFF0000h	Unknown system error (miscellaneous error).
VI_ERROR_INV_OBJECT	BFFF000Eh	The given session or object reference is invalid.
VI_ERROR_RSRC_LOCKED	BFFF000Fh	Specified type of lock cannot be obtained or specified operation cannot be performed, because the resource is locked.
VI_ERROR_INV_EXPR	BFFF0010h	Invalid expression specified for search.
VI_ERROR_RSRC_NFOUND	BFFF0011h	Insufficient location information or the device or resource is not present in the system.
VI_ERROR_INV_RSRC_NAME	BFFF0012h	Invalid resource reference

specified. Parsing error. BFFF0013h Invalid access VI\_ERROR\_INV\_ACC\_MODE mode. VI\_ERROR\_TMO **BFFF0015h** Timeout expired before operation completed. BFFF0016h Unable to VI\_ERROR\_CLOSING\_FAILED deallocate the previously allocated data structures corresponding to this session or object reference. **BFFF001Bh** Specified VI ERROR INV DEGREE degree is invalid. VI ERROR INV JOB ID BFFF001Ch Specified job identifier is invalid. VI\_ERROR\_NSUP\_ATTR BFFF001Dh The specified attribute is not defined or supported by the referenced session, event, or find list. VI\_ERROR\_NSUP\_ATTR\_STATE **BFFF001Eh** The specified state of the attribute is not valid, or is not supported as

defined by the

		session, event, or find list.
VI_ERROR_ATTR_READONLY	BFFF001Fh	The specified attribute is Read Only.
VI_ERROR_INV_LOCK_TYPE	BFFF0020h	The specified type of lock is not supported by this resource.
VI_ERROR_INV_ACCESS_KEY	BFFF0021h	The access key to the resource associated with this session is invalid.
VI_ERROR_INV_EVENT	BFFF0026h	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	BFFF0027h	Invalid mechanism specified.
VI_ERROR_HNDLR_NINSTALLED	BFFF0028h	A handler is not currently installed for the specified event.
VI_ERROR_INV_HNDLR_REF	BFFF0029h	The given handler reference is invalid.
VI_ERROR_INV_CONTEXT	BFFF002Ah	Specified event context is invalid.
VI_ERROR_QUEUE_OVERFLOW	BFFF002Dh	The event queue for the specified type

		has overflowed (usually due to previous events not having been closed).
VI_ERROR_NENABLED	BFFF002Fh	The session must be enabled for events of the specified type in order to receive them.
VI_ERROR_ABORT	BFFF0030h	The operation was aborted.
VI_ERROR_RAW_WR_PROT_VIOL	BFFF0034h	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	BFFF0035h	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	BFFF0036h	Device reported an output protocol error during transfer.
VI_ERROR_INP_PROT_VIOL	BFFF0037h	Device reported an input protocol error during transfer.
VI_ERROR_BERR	BFFF0038h	Bus error occurred during transfer.
VI_ERROR_IN_PROGRESS	BFFF0039h	Unable to queue the asynchronous

		operation because there is already an operation in progress.
VI_ERROR_INV_SETUP	BFFF003Ah	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_QUEUE_ERROR	BFFF003Bh	Unable to queue asynchronous operation (usually due to the I/O completion event not being enabled or insufficient space in the session's queue).
VI_ERROR_ALLOC	BFFF003Ch	Insufficient system resources to perform necessary memory allocation.
VI_ERROR_INV_MASK	BFFF003Dh	Invalid buffer mask specified.
VI_ERROR_IO	BFFF003Eh	Could not perform

operation because of I/O error. VI\_ERROR\_INV\_FMT BFFF003Fh A format specifier in the format string is invalid. BFFF0041h A format VI\_ERROR\_NSUP\_FMT specifier in the format string is not supported. VI\_ERROR\_LINE\_IN\_USE BFFF0042h The specified trigger line is currently in use. BFFF0046h The specified VI ERROR NSUP MODE mode is not supported by this VISA implementation. **BFFF004Ah** Service request VI ERROR SRQ NOCCURRED has not been received for the session. **BFFF004Eh** Invalid address VI ERROR INV SPACE space specified. VI\_ERROR\_INV\_OFFSET BFFF0051h Invalid offset specified. VI ERROR INV WIDTH BFFF0052h Invalid source or destination width specified. BFFF0054h Specified offset VI ERROR NSUP OFFSET is not accessible from this hardware.

VI_ERROR_NSUP_VAR_WIDTH	BFFF0055h	Cannot support source and destination widths that are different.
VI_ERROR_WINDOW_NMAPPED	BFFF0057h	The specified session is not currently mapped.
VI_ERROR_RESP_PENDING	BFFF0059h	A previous response is still pending, causing a multiple query error.
VI_ERROR_NLISTENERS	BFFF005Fh	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_NCIC	BFFF0060h	The interface associated with this session is not currently the controller in charge.
VI_ERROR_NSYS_CNTLR	BFFF0061h	The interface associated with this session is not the system controller.
VI_ERROR_NSUP_OPER	BFFF0067h	The given session or object reference does not support this

		operation.
VI_ERROR_INTR_PENDING	BFFF0068h	An interrupt is still pending from a previous call.
VI_ERROR_ASRL_PARITY	BFFF006Ah	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	BFFF006Bh	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	BFFF006Ch	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_TRIG_NMAPPED	BFFF006Eh	The path from trigSrc to trigDest is not currently mapped.
VI_ERROR_NSUP_ALIGN_OFFSET	BFFF0070h	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_USER_BUF	BFFF0071h	A specified user buffer is not valid or cannot be accessed for

		the required size.
VI_ERROR_RSRC_BUSY	BFFF0072h	The resource is valid, but VISA cannot currently access it.
VI_ERROR_NSUP_WIDTH	BFFF0076h	Specified width is not supported by this hardware.
VI_ERROR_INV_PARAMETER	BFFF0078h	The value of some parameter— which parameter is not known—is invalid.
VI_ERROR_INV_PROT	BFFF0079h	The protocol specified is invalid.
VI_ERROR_INV_SIZE	BFFF007Bh	Invalid size of window specified.
VI_ERROR_WINDOW_MAPPED	BFFF0080h	The specified session currently contains a mapped window.
VI_ERROR_NIMPL_OPER	BFFF0081h	The given operation is not implemented.
VI_ERROR_INV_LENGTH	BFFF0083h	Invalid length specified.
VI_ERROR_INV_MODE	BFFF0091h	The specified

		mode is invalid.
VI_ERROR_SESN_NLOCKED	BFFF009Ch	The current session did not have any lock on the resource.
VI_ERROR_MEM_NSHARED	BFFF009Dh	The device does not export any memory.
VI_ERROR_LIBRARY_NFOUND	BFFF009Eh	A code library required by VISA could not be located or loaded.
VI_ERROR_NSUP_INTR	BFFF009Fh	The interface cannot generate an interrupt on the requested level or with the requested statusID value.
VI_ERROR_INV_LINE	BFFF00A0h	The value specified by the line parameter is invalid.
VI_ERROR_FILE_ACCESS	BFFF00A1h	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
VI_ERROR_FILE_IO	BFFF00A2h	An error

		occurred while performing I/O on the specified file.
VI_ERROR_NSUP_LINE	BFFF00A3h	One of the specified lines (trigSrc or trigDest) is not supported by this VISA implementation, or the combination of lines is not a valid mapping.
VI_ERROR_NSUP_MECH	BFFF00A4h	The specified mechanism is not supported for the given event type.
VI_ERROR_INTF_NUM_NCONFIG	BFFF00A5h	The interface type is valid but the specified interface number is not configured.
VI_ERROR_CONN_LOST	BFFF00A6h	The connection for the given session has been lost.
VI_ERROR_MACHINE_NAVAIL	BFFF00A7h	The remote machine does not exist or is not accepting any connections.
VI_ERROR_NPERMISSION	BFFF00A8h	Access to the

resource or remote machine is denied. This is due to lack of sufficient privileges for the current user or machine.

## **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 and GPIB-VXI 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 and GPIB-VXI 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.

**Programming Considerations** 

NI Spy: Debugging Tool

Multiple Applications Using the NI-VISA Driver

Low-Level Access Functions

**Interrupt Callback Handlers** 

Multiple Interface Support Issues

VXI and GPIB Platforms

Serial Port Support

VME Support

# **Programming Considerations**

The following topics contain information for you to consider when developing applications that use the NI-VISA I/O interface software.

NI Spy: Debugging Tool Multiple Applications Using the NI-VISA Driver Low-Level Access Functions Interrupt Callback Handlers

# **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 viPeek*XX*() or viPoke*XX*(). 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 such as the GPIB-VXI, you must use the viPeek*XX*() and viPoke*XX*() 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 viPeek*XX*() and viPoke*XX*() 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**

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 (32bit). 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).

When using National Instruments Measurement Studio for Visual C++, callbacks are registered with the InstallEventHandler() method. See 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)

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 way VISA invokes callbacks is platform dependent, as shown in the following table.

Platform	Callback Invocation Method
Windows Vista/XP/2000, Linux <i>x</i> 86, Mac OS X, LabVIEW RT	The callback is performed in a separate thread created by NI-VISA. The thread is signaled as soon as the event occurs.
VxWorks <i>x</i> 86	For VXI, the callback is performed from within the driver interrupt service routine. For all other interfaces, the callback is performed only when the driver is accessed.

#### How VISA Invokes Callbacks

What this means is that on VxWorks (all interfaces other than VXI) you

cannot wait in a tight loop for a callback to occur. For example, the following code does not work:

```
while (!intr_recv)
; /* do nothing */
```

For callbacks to be invoked on the VxWorks platform, you must call any VISA operation or give up processor time. Notice that NI-VISA on Windows and all UNIX platforms does not require you to call VISA operations or give up processor time to receive callbacks. However, because occasionally calling VISA operations ensures that callbacks will be invoked correctly on any platform, you should keep these issues in mind when writing code that you want to be portable.

## **Multiple Interface Support Issues**

This section contains information about how to use or configure your NI-VISA software for certain types of interfaces.

VXI and GPIB Platforms

Serial Port Support

VME Support

# **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, GPIB-VXI, 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. With the GPIB-ENET and GPIB-ENET/100 boxes, you can even remotely control VXI devices when using a GPIB-VXI controller.

# **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.

Platform	Method
Windows Vista/XP/2000	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 <i>x</i> 86	ASRL1-ASRL4 access /dev/ttyS0 – /dev/ttyS3.
VxWorks x86	ASRL1-ASRL2 access /tyCo/0 – /tyCo/1.

#### How Serial Ports Are Numbered

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.

### Example

```
\overline{\mathbb{N}}
```

**Note** This example shows C source code. There is also an example in <u>Visual Basic</u> syntax.

```
#include "visa.h"
int main(void)
```

```
{
```

```
/* For checking errors */
ViStatus
            status:
                                                /* Communication channel
ViSession
            defaultRM;
ViSession
                                                /* Communication channel
            instr:
ViChar
            rsrcName[VI FIND BUFLEN];
                                               /* Serial resource name */
ViChar
            intfDesc[VI FIND BUFLEN];
                                               /* Port binding description
ViUInt32
            retCount:
                                                /* To hold number of resou
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, rsrcN
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;
```

}

## Serial Port Support Example (Visual Basic)

Note The Visual Basic examples in the *NI-VISA Help* 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 Declare Function viGetAttrString Lib "VISA32.DLL" Alias "#133" (By As ViSession, ByVal attrName As ViAttr, ByVal strValue As Any) As ViStatus

Private Sub vbMain()

Dim statAs ViStatusDim dfltRMAs ViSessionDim sesnAs ViSessionDim fListAs ViFindListDim rsrcNameAs String \* VI\_FIND\_BUFLENDim instrDescAs String \* VI\_FIND\_BUFLENDim nListAs Longstat = viOpenD=taultRM(dfltRM)If (stat < VI\_SUCCESS) Then</td>

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</pre>
```

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 blockmove operations, as well as the ability to receive interrupts.



#### Prefixes

Symbol	Prefix	Value
р	pico	10 -12
n	nano	10 <sup>-9</sup>
μ	micro	10 -6
m	milli	10 <sup>-3</sup>
k	kilo	10 <sup>3</sup>
М	mega	10 6
G	giga	10 <sup>9</sup>
Т	tera	10 12

## Numbers/Symbols

nV	nanovolts	10 <sup>-9</sup> volts
μV	microvolts	10 <sup>-6</sup> volts
μΩ	microohms	10 <sup>-6</sup> ohms
mΩ	milliohms	10 <sup>-3</sup> ohms
MΩ	megaohms	10 <sup>6</sup> ohms
pА	picoamps	10 <sup>-12</sup> amperes
nA	nanoamps	10 <sup>-9</sup> amperes
μA	microamps	10 <sup>-6</sup> amperes
mA	milliamps	10 <sup>-3</sup> amperes

## Α

address	A string (or other language construct) that uniquely locates and identifies a resource. VISA defines an ASCII- based grammar that associates strings with particular physical devices and VISA resources.
address location	Refers to the location of a specific register.
address modifier	One of six signals in the VMEbus specifications used by VMEbus masters to indicate the address space and mode (supervisory/nonprivileged, data/program/block) in which a data transfer is to take place.
address space	In VXI/VME systems, a set of 2 <sup><i>n</i></sup> memory locations differentiated from other such sets in VXI/VMEbus systems by six signal lines known as address modifiers, where <i>n</i> (either 16, 24, or 32) is the number of address lines required to uniquely specify a byte location in a given space. In PXI systems, the address space corresponds to 1 of 6 possible BAR locations (BAR0 through BAR5). In VME, VXI, and PXI, a given device may have addresses in one or more address spaces.
address string	A string (or other language construct) that uniquely locates and identifies a resource. VISA defines an ASCII- based grammar that associates strings with particular physical devices and VISA resources.
alias	User-defined name for a VISA resource.
ANSI	American National Standards Institute
API	Application Programming Interface. The direct interface that an end user sees when creating an application. In VISA, the API consists of the sum of all of the operations, attributes, and events of each of the VISA Resource Classes.
ASCII	American Standard Code for Information Interchange.
asynchronous	An action or event that occurs at an unpredictable time with respect to the execution of a program.

attribute A value within an object or resource that reflects a characteristic of its operational state.

b Bit

B Byte

backplane In VXI/VME systems, an assembly, typically a PCB, with 96pin connectors and signal paths that bus the connector pins. A C-size VXIbus system will have two sets of bused connectors called the J1 and J2 backplanes. A D-size VXIbus system will have three sets of bused connectors called the J1, J2, and J3 backplane.

Base Each PCI or PXI device has six of these, BAR0 through

- Address BAR5. At power-on, each BAR requests a given size of Register memory or I/O space. Each device can request from 0 to 6 regions of PCI memory or I/O space. After the operating system starts, each BAR contains an assigned base address in PCI address space. A value of 0 in a given BAR indicates that the device is not using that BAR.
- bus error An error that signals failed access to an address. Bus errors occur with low-level accesses to memory and usually involve hardware with bus mapping capabilities. For example, nonexistent memory, a nonexistent register, or an incorrect device access can cause a bus error.
- byte order How bytes are arranged within a word or how words are arranged within a longword. Motorola (Big-Endian) ordering stores the most significant byte (MSB) or word first, followed by the least significant byte (LSB) or word. Intel (Little-Endian) ordering stores the LSB or word first, followed by the MSB or word.

В

## С

callback	See also handler. A software routine that is invoked when an asynchronous event occurs. In VISA, callbacks can be installed on any session that processes events.
CIC	Controller-In-Charge. The device that manages the GPIB by sending interface messages to other devices.
commander	A device that has the ability to control another device. This term also can denote the unique device that has sole control over another device (as with the VXI Commander/Servant hierarchy).
communication channel	The same as session. A communication path between a software element and a resource. Every communication channel in VISA is unique.
configuration registers	A set of registers through which the system can identify a module device type, model, manufacturer, address space, and memory requirements. In order to support automatic system and memory configuration, the PXI and VXIbus specifications require that all PXI and VXIbus devices have a set of such registers.
controller	An entity that can control another device(s) or is in the process of performing an operation on another device.
CPU	Central processing unit

### D

- device An entity that receives commands from a controller. A device can be an instrument, a computer (acting in a non-controller role), or a peripheral (such as a plotter or printer).
- DLL Dynamic Link Library. See also a shared library or shared object. A file containing a collection of functions that can be used by multiple applications. This term is usually used for libraries on Windows platforms.
- DMA Direct memory access. High-speed data transfer between a board and memory that is not handled directly by the CPU. Not available on some systems. See programmed I/O.

### Ε

embedded A computer plugged directly into the VXI backplane. An controller example is the National Instruments VXIpc-870.

event An asynchronous occurrence that is independent of the normal sequential execution of the process running in a system.

external A desktop computer or workstation connected to the VXI system via a MXI interface board. An example is a standard personal computer with a PCI-MXI-2 installed.

## F

Fast See <u>FDC</u>.

Data Channel

- FDC Fast Data Channel; a protocol that provides a mechanism for transferring data blocks between a VXIbus Commander and its Servants.
- FIFO First In-First Out; a method of data storage in which the first element stored is the first one retrieved.

### G

global A global attribute is one whose value is the same for all attribute sessions to the specified resource. An example of this is the hardware interface type.

GPIB General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1992.

## Η

- handler See also callback. A software routine that is invoked when an asynchronous event occurs. In VISA, callbacks can be installed on any session that processes events.
- handshaking A type of protocol that makes it possible for two devices to synchronize operations.

I	
I/O	input/output
IEEE	Institute of Electrical and Electronics Engineers
instrument	A device that accepts some form of stimulus to perform a designated task, test, or measurement function. Two common forms of stimuli are message passing and register reads and writes. Other forms include triggering or varying forms of asynchronous control.
instrument driver	A set of routines designed to control a specific instrument or family of instruments, and any necessary related files for LabWindows/CVI or LabVIEW.
interface	A generic term that applies to the connection between devices and controllers. It includes the communication media and the device/controller hardware necessary for cross-communication.
interrupt	A condition that requires attention out of the normal flow of control of a program.
IVI	Interchangeable Virtual Instruments
IVI Driver	A software module that controls a hardware device and that complies with the IVI Foundation specifications.
IVI Foundation, Inc.	Interchangeable Virtual Instruments, Inc., a non-profit Delaware Corporation, composed of end-user test engineers, instrument and software suppliers, and system integrators, chartered to define software standards that promote instrument interchangeability. See www.ivifoundation.org for more details.

## L

local A local attribute is one whose value is unique to the session. attribute An example of this is the timeout.

- lock A state that prohibits sessions other than the session(s) owning the lock from accessing a resource.
- logical An 8-bit number that uniquely identifies the location of each address VXIbus device's configuration registers in a system. The A16 register address of a device is C000h + Logical Address \* 40h.

#### Μ

- mapping An operation that returns a reference to a specified section of an address space and makes the specified range of addresses accessible to the requester. This function is independent of memory allocation.
- MAX Measurement & Automation Explorer. Provides access to all National Instruments DAQ, GPIB, IMAQ, IVI, Motion, VISA, and VXI devices. With MAX, you can configure National Instruments 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.
- In VXI/VME systems, an intelligent device that implements the defined VXIbus registers and communication protocols. These devices are able to use Word Serial Protocol to communicate with one another through communication registers. All GPIB and Serial devices are by definition message-based, as are devices for some other interfaces. Many modern message-based devices support the IEEE 488.2 protocol.
- multitasking The ability of a computer to perform two or more functions simultaneously without interference from one another. In operating system terms, it is the ability of the operating system to execute multiple applications/processes by timesharing the available CPU resources.

### Ν

NI A utility that monitors, records, and displays multiple National Spy Instruments APIs, such as NI-488.2 and NI-VISA. Useful for troubleshooting errors in your application and for verifying communication. 0

operation An action defined by a resource that can be performed on a resource. In general, this term is synonymous with the connotation of the word method in object-oriented architectures.

### Ρ

- process An operating system element that shares a system's resources. A multi-process system is a computer system that allows multiple programs to execute simultaneously, each in a separate process environment. A single-process system is a computer system that allows only a single program to execute at a given point in time.
- programmed Low-speed data transfer between a board and memory in I/O which the CPU moves each data value according to program instructions. See DMA.
- protocol Set of rules or conventions governing the exchange of information between computer systems.
- PXI PCI eXtensions for Instrumentation. PXI leverages the electrical features defined by the Peripheral Component Interconnect (PCI) specification as well as the CompactPCI form factor, which combines the PCI electrical specification with Eurocard (VME) mechanical packaging and high-performance connectors. This combination allows CompactPCI and PXI systems to have up to seven peripheral slots versus four in a desktop PCI system.

## R

register An address location that can be read from or written into or both. It may contain a value that is a function of the state of hardware or can be written into to cause hardware to perform a particular action. In other words, an address location that controls and/or monitors hardware.

registerbased In VXI/VME systems, a servant-only device that supports only the four basic VXIbus configuration registers. Register-based device devices are typically controlled by message-based devices via device-dependent register reads and writes. All PXI devices are by definition register-based, as are devices for some other interfaces.

Resource The definition for how to create a particular resource. In Class general, this is synonymous with the connotation of the word class in object-oriented architectures. For VISA Instrument Control resource classes, this refers to the definition for how to create a resource which controls a particular capability or set of capabilities of a device.

resource In general, this term is synonymous with the connotation of or the word object in object-oriented architectures. For VISA, resource more specifically refers to a particular instance implementation (or instance in object-oriented terms) of a Resource Class.

#### s second

- SCPI Standard Commands for Programmable Instrumentation; a protocol which defines a standard set of commands to control programmable test and measurement devices in instrumentation systems.
- servant A device controlled by a Commander.
- session The same as communication channel. A communication path between a software element and a resource. Every communication channel in VISA is unique.

shared See also DLL. A file containing a collection of functions that library or can be used by multiple applications. This term is usually used

shared for libraries on UNIX platforms. object

- shared A block of memory that is accessible to both a client and a
- memory server. The memory block operates as a buffer for communication. This is unique to register-based interfaces such as VXI.
- socket A bi-directional communication endpoint; an object through which a VISA sockets application sends or receives packets of data across a network.
- SRQ IEEE 488 Service Request. This is an asynchronous request from a remote device that requires service. A service request is essentially an interrupt from a remote device. For GPIB, this amounts to asserting the SRQ line on the GPIB. For VXI, this amounts to sending the Request for Service True event (REQT).
- status A byte of information returned from a remote device that byte shows the current state and status of the device. If the device follows IEEE 488 conventions, bit 6 of the status byte indicates whether the device is currently requesting service.
- status/ID A value returned during an IACK cycle. In VME, usually an 8bit value which is either a status/data value or a vector/ID value used by the processor to determine the source. In VXI,

S

a 16-bit value used as a data; the lower 8 bits form the VXI logical address of the interrupting device and the upper 8 bits specify the reason for interrupting.

## Т

- TCP/IP Transmission Control Protocol/Internet Protocol. The de facto standard for transmitting data over networks, TCP/IP is a multilayered suite of communication protocols used to connect hosts on LANs, WANs and the Internet. It is very widely supported, even by network operating systems that have their own communication protocols.
- thread An operating system element that consists of a flow of control within a process. In some operating systems, a single process can have multiple threads, each of which can access the same data space within the process. However, each thread has its own stack and all threads can execute concurrently with one another (either on multiple processors, or by time-sharing a single processor).

U

USB Universal Serial Bus. This common computer peripheral bus allows up to 127 individual USB peripherals per USB root node.

#### V

virtual A name given to the grouping of software modules (in this instrument case, VISA resources with any associated or required hardware) to give the functionality of a traditional standalone instrument. Within VISA, a virtual instrument is the logical grouping of any of the VISA resources.

VISA Virtual Instrument Software Architecture. This is the general name given to this product and its associated architecture. The architecture consists of two main VISA components: the VISA resource manager and the VISA resources.

VISA This is the name given to the part of VISA that defines all of Instrument the device-specific resource classes. VISA Instrument Control Control resources encompass all defined device capabilities Resources for direct, low-level instrument control.

VISA This is the name given to the part of VISA that defines all of the register or memory-specific resource classes. The VISA MEMACC resources encompass all high and low-level services for interface-level accesses to all memory defined in the system.

VISA This is the name given to the part of VISA that manages Resource resources. This management includes support for finding Manager resources and opening sessions to them.

VISA This is the name given to the part of VISA that defines the basic constraints and interface definition for the creation and use of a VISA resource. All VISA resources must derive their interface from the definition of the VISA Resource Template. This includes services for setting and retrieving attributes, receiving events, locking resources, and closing objects.

visaconf VISA configuration utility for Linux and Mac OS X.

VISAIC VISA Interacvtive Control utility. Interactively controls VXI/VME devices without using a conventional programming language, LabVIEW, or Measurement Studio.

VME Versa Module Eurocard or IEEE 1014

VXIbus VMEbus Extensions for Instrumentation or IEEE 1155

# **Important Information**

Warranty Copyright Trademarks Patents Warning Regarding Use of NI Products

## Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

# Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

# Trademarks

National Instruments, NI, ni.com, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on ni.com/legal for more information about <u>National Instruments trademarks</u>.

FireWire® is the registered trademark of Apple Computer, Inc.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler™ are trademarks of The MathWorks, Inc.

Tektronix® and Tek are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

## Patents

For patents covering National Instruments products, refer to the appropriate location: **HelpPatents** in your software, the patents.txt file on your media, or <u>ni.com/patents</u>.

## WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS. SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE. INJURY. OR DEATH. THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES. INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR

APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
## **Technical Support and Professional Services**

Visit the following sections of the award-winning National Instruments Web site at ni.com for technical support and professional services:

- <u>Support</u>—Technical support resources at ni.com/support include the following:
  - Self-Help Resources—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-bystep troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - Standard Service Program Membership—This program entitles members to direct access to NI Applications Engineers via phone and email for one-toone technical support, as well as exclusive access to on demand training modules via the <u>Services Resource</u> <u>Center</u>. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

For information about other <u>technical support options</u> in your area, visit ni.com/services or <u>contact</u> your local office at ni.com/contact.

- <u>Training and Certification</u>—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- <u>System Integration</u>—If you have time constraints, limited inhouse technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your <u>local office</u> or NI corporate headquarters. You also can visit the <u>Worldwide Offices</u> section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

# **Unit Specification ID**

This identifies the specification to which the FireWire device adheres.

### **Unit Software Version**

This identifies the specification version to which the FireWire device adheres.

## **Manufacturer Name**

The device manufacturer name.

### **Model Name**

The FireWire module name.

### **Manufacturer Code**

The 16-bit PCI Vendor ID for this device. A list of PCI Vendor ID values is maintained by the PCI Special Interest Group (SIG).

## **Manufacturer Name**

The device manufacturer name.

### Model Code

The 16-bit PCI Device ID for this device. The Device ID is uniquely assigned to a PXI/PCI module by the instrument vendor.

### **Model Name**

The PXI/PCI module name.

# **Generates Interrupts**

Check this box if your PXI/PCI device is capable of asserting interrupts and you would like to be able to receive interrupt notification using NI-VISA.

### Subsystem Manufacturer Code

The 16-bit PCI Vendor ID used in the subsystem for this device. The PCI Special Interest Group (SIG) maintains a list of PCI Vendor ID values.

## Subsystem Model Code

The 16-bit PCI Device ID used in the subsystem for this device. The instrument vendor uniquely assigns the Device ID to a PXI/PCI module.

# **Device Uses Subsystem**

This device has defined values for the PCI subsystem Vendor ID and Device ID.

### 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.

# Load settings from Module Description file...

If a Module Description file (also called a module.ini file) is available for this device, you can import the information from that file into the wizard instead of entering the settings yourself. For more information about Module Description files, click the **Help** button.

## More about Module Description Files

Module Description Files (also called module.ini files) provide a mechanism for informing the operating system and the PXI Resource Manager about key attributes of a PXI module. The module description file accomplishes this with two primary features:

- It specifies how the operating system should identify this device and associate it with the VISA implementation.
- It specifies how the PXI Resource Manager should handle combination modules with multiple PCI devices and/or multiple PCI functions.

A module vendor provides a module description file to enable a user to easily configure the product into a PXI system. The specification for these files is at <u>http://www.pxisa.org/Specifications.html</u>.

The NI-VISA Driver Wizard uses module.ini files to allow you to easily load settings for a given VISA PXI/PCI instrument. The wizard generates a module.ini file when it creates its output files, and you can reload this file later to update settings used by the wizard. You can also distribute the module.ini file to users of your instrument.



**Note** The wizard currently reads, but does not write, module.ini files describing combination (multifunction or multidevice) modules in a single module.ini file.

### **Manufacturer Code**

The 16-bit USB Vendor ID (VID) for this device. A list of USB Vendor ID values is maintained by the USB Implementers Forum (USB-IF).

### **Model Code**

The 16-bit USB Product ID (PID) for this device. The Product ID is uniquely assigned to a USB module by the instrument vendor.

### **Model Name**

The USB module name.

# **Compound Device**

A USB compound device is a device that supports two or more USB interfaces.

### **Interrupt Detection Background**

Because PCI devices share one of four physical interrupt lines, it is possible that more than one PXI/PCI device can be interrupting at any given time. In the Interrupt Detection dialog, you specify the sequence of register operations that allow NI-VISA to determine whether or not your device is interrupting. PCI hardware typically indicates a pending interrupt condition using an Interrupt Status/Control register.

For the purposes of determining whether or not your device is asserting a hardware interrupt, a Read/Compare operation exists. This operation will perform a register read, applying a user-defined mask (logical-AND) to the contents of the register. The resulting value is then compared with a user-specified constant (using another logical-AND). If the masked-result and the user-defined constant are the same, the comparison operation is said to be *True*. If the values are different, the result is *False*. If the result of all Read/Compare operations in a sequence of register transactions is *True*, NI-VISA will conclude that your device is interrupting and proceed to execute the Interrupt Acknowledge sequence. Because NI-VISA relies on the result of the comparison operations in making this conclusion, at least one Read/Compare operation must be present in this transaction sequence.

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, if any of them have a result of *True*, NI-VISA concludes that this interrupt belongs to this device.

# Add a step before

Adds a register operation prior to the currently selected step.

# Add a step after

Adds a register operation immediately following the currently selected step.

## Edit a step

Allows you to modify the properties of the currently selected register operation.

### Remove a step

Removes the currently selected step from the sequence of register operations.

## Select sequence

Selects a transaction sequence to view or modify.

## Add sequence

Adds a new transaction sequence.

### **Remove sequence**

Removes the currently selected transaction sequence.

### **Interrupt Removal Background**

On modern operating systems, hardware interrupts are typically serviced in a high-priority execution context that runs as part of the OS kernel. When a hardware interrupt occurs, the OS kernel will execute its chain of interrupt service routines until a single ISR "claims" the interrupt by telling the OS kernel that its device is currently interrupting. Once the interrupt is claimed, the individual ISR must acknowledge the pending interrupt before returning control to the operating system. When the interrupt is no longer pending, the OS will return control to the user application (for example, your instrument driver). By specifying the steps necessary to acknowledge a pending interrupt for your device, you are telling NI-VISA how to service interrupts for your device inside the OS kernel.

# Type of access

Register accesses assume the form of Reads, Writes, or Compares:

**Read**: Performs a register read (of specified width) from a given offset relative to a given address space.

**Write**: Performs a register write of a given value (of specified width) to a given offset relative to a given address space.

**Compare**: Performs a Read operation, but additionally applies a userdefined mask to the result of the read operation (using a logical AND). The result of the bit mask is then compared to another user supplied value. The Compare operation is useful for examining individual bits or combinations of bits within a single register.

### **Address space**

The PCI Address space that this operation applies to. Valid address spaces include the PCI Configuration address space (CFG), and any of the Memory or IO address spaces defined in your device's base address registers (BAR0-BAR5).

### **Compare mask**

The mask that will be applied to the result of the register read in a Compare operation. The mask will be logically AND'd with the value read from the register. This field is only valid when a Compare access type is specified.

### Width of access

The width (in bits) of the register operation.

# Offset within space

The offset within the specified address space to perform this register operation.

### Value to write or compare

For a Write operation, this is the value to be written to the register. For a Compare operation, this is the value to compare the masked result with.
# **Output Files Background**

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

The NI-VISA Driver Wizard will generate these Setup Information files for you. The format and handling of each file varies depending on the operating system version. The NI-VISA Driver Wizard generates .inf files for the following operating systems depending on the selected bus:

#### PXI/PCI

- Windows Vista/XP/2000
- LabVIEW RT
- Mac OS X
- Linux

#### USB

• Windows Vista/XP/2000

#### **FireWire**

- Windows Vista/XP/2000
- LabVIEW RT

When the operating system scans the hardware bus, if a new piece of hardware is detected, the OS will compare that device's characteristics with the contents of the <Windows>\inf directory. For PXI/PCI devices, the hardware bus is scanned by the operating system at boot time. For USB devices, the hardware bus is scanning whenever a device is plugged in. If a .inf file is found that describes the newly detected hardware, the OS will execute the driver-specific portion of the .inf file (for example, adding registry entries and copying files).

## **Instrument Prefix**

The VXIplug&play-compliant instrument driver prefix for this device. The instrument prefix is used in forming the names of the output files that the wizard will generate. The resulting filenames are consistent with instrument driver files created with LabWindows/CVI. If you are not creating a VXIplug&play driver, and you simply need to use VISA to access your device, this field can be left in its default state.

# **Output File Directory**

The directory where the output files should be saved. By default, the wizard will attempt to save the files in the instrument driver directory for this device (based on the Instrument Prefix).

On Windows, the default is:

My Documents\National Instruments\NI-VISA\<Instrument Prefix>

On Linux, the default is:

<User Home>/natinst/NI-VISA/<Instrument Prefix>

On Mac OS X, the default is:

<User Home>/Documents/National Instruments/NI-VISA/<Instrument Prefix>

LabWindows/CVI will look for the files in this location when the Create Distribution Kit option is used to build an instrument driver installation package. Currently, LabWindows/CVI supports distribution of .inf files for PXI/PCI, USB, and FireWire devices.

## What Do I Do with the Output Files?

The procedure for handling the Setup Information files is OS dependent. If you are using LabWindows/CVI to build your instrument driver, the distribution kit handles the multiple-OS issues for you.

Using LabWindows/CVI to Install Your Device .inf Files

If you are creating your own installation package...

# Install the Setup Information file

This option installs the corresponding Setup Information (. $\inf$ ) file to the local system.

# FTP to LabVIEW RT system

This option allows you to download your device's .inf file to the target system. This option is available only if LabVIEW RT is installed and you are generating Setup Information (.inf) files for PXI/PCI or FireWire.

# Go to folder

This option opens the folder that contains the generated Setup Information (.inf) files.

# Do nothing more

This option exits the wizard without performing any additional tasks.

## **Branch Offices**

Office	Telephone Number
Australia	1800 300 800
Austria	43 662 457990-0
Belgium	32 (0) 2 757 0020
Brazil	55 11 3262 3599
Canada	800 433 3488
China	86 21 5050 9800
Czech Republic	420 224 235 774
Denmark	45 45 76 26 00
Finland	358 (0) 9 725 72511
France	33 (0) 1 57 66 24 24
Germany	49 89 7413130
India	91 80 41190000
Israel	972 0 3 6393737
Italy	39 02 41309277
Japan	0120-527196 / 81 3 5472 2970
Korea	82 02 3451 3400
Lebanon	961 (0) 1 33 28 28
Malaysia	1800 887710
Mexico	01 800 010 0793
Netherlands	31 (0) 348 433 466
New Zealand	0800 553 322
Norway	47 (0) 66 90 76 60
Poland	48 22 3390150
Portugal	351 210 311 210
Russia	7 495 783 6851
Singapore	1800 226 5886
Slovenia	386 3 425 42 00

South Africa	27 0 11 805 8197
Spain	34 91 640 0085
Sweden	46 (0) 8 587 895 00
Switzerland	41 56 2005151
Taiwan	886 02 2377 2222
Thailand	662 278 6777
Turkey	90 212 279 3031
United Kingdom	44 (0) 1635 523545
United States (Corporate)	512 683 0100

# If You Are Creating Your Own Installation Package...

**Windows XP/2000**: Copy the .inf file to the <Windows>\inf directory and reboot the computer. The OS Plug&Play manager will associate the new hardware device with the NI-VISA driver once the .inf file is in place. If the hardware was installed before the driver software and .inf file, the device may need to be removed from the Device Manager before rebooting.

**Windows Vista**: In Windows Explorer, right-click the .inf file and choose **Install**. If you plugged in the device before installing this file, you may need to remove the device from the **Unknown Devices** class in the Windows Device Manager.

**Linux**: (PXI/PCI Only) Copy the .inf file to /etc/natinst/nipal/inf and reboot the computer.

**Mac OS X**: (PXI/PCI Only) Copy the .inf file to /Library/Application Support/National Instruments/nipal/inf and reboot the computer.