VISA

NI-VISA™ User Manual

NATIONAL INSTRUMENTS™

June 1998 Edition
Part Number 321074D-01

Internet Support
E-mail: `support@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `www.natinst.com`

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway   Austin, Texas 78730-5039   USA   Tel: 512 794 0100

Important Information

Copyright

Trademarks

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

Contents

Chapter 6
Register-Based Communication

Chapter 7
VISA Events

Chapter 8
VISA Locks

About This Manual

Organization of This Manual

The *NI-VISA User Manual* is organized as follows:

- Chapter 1, *Introduction*, discusses how to use this manual, lists what you need to get started, and contains a brief description of the VISA Library.
- Chapter 2, *Introductory Programming Examples*, introduces some examples of common communication with instruments.
- Chapter 3, *VISA Overview*, contains an overview of the VISA Library.
- Chapter 4, *Initializing Your VISA Application*, describes the steps required to prepare your application for communication with your device.
- Chapter 5, *Message-Based Communication*, shows how to use the VISA library in message-based communication.
- Chapter 6, *Register-Based Communication*, shows how to use the VISA library in register-based communication.
- Chapter 7, *VISA Events*, describes the VISA event model and how to use it.
- Chapter 8, *VISA Locks*, describes how to use locks in VISA.
- Chapter 9, *NI-VISA Platform-Specific and Portability Issues*, discusses programming information for you to consider when developing applications that use the NI-VISA driver.
- Appendix A, *Visual Basic Examples*, shows the Visual Basic syntax of the ANSI C examples given earlier in this manual. The examples use the same numbering sequence for easy reference.
- Appendix B, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. For example, the sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.
♦	The ♦ symbol indicates that the text following it applies only to a specific product or a specific operating system.
☞	This icon to the left of bold italicized text denotes a note, which alerts you to important information.
bold	Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, windows, or Windows 95 tabs.
bold italic	Bold italic text denotes a note, caution, or warning.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.
`monospace`	Text in this font denotes text or characters that you should literally 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, and for statements and comments taken from programs.
`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 that you must enter the appropriate words or values in the place of these items.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

How to Use This Document Set

Use the documentation that came with your GPIB and/or VXI hardware and software for Windows to install and configure your system.

Refer to the Read Me First document for information on installing the NI-VISA distribution media.

Use the *NI-VISA User Manual* for detailed information on how to program using VISA.

Use the NI-VISA online help or the *NI-VISA Programmer Reference Manual* for specific information about the attributes, events, and operations, such as format, syntax, parameters, and possible errors.

♦ **Windows 95/NT users**—The *NI-VISA Programmer Reference Manual* is not included in Windows 95/NT kits. Windows 95/NT users can access this information through the `NI-visa.hlp` file at **Start»Programs»VXIpnp»VISA Help**.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- 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*
- *NI-488.2 Function Reference Manual for DOS/Windows*, National Instruments Corporation
- *NI-488.2 User Manual for Windows*, National Instruments Corporation
- *NI-VXI Programmer Reference Manual*, National Instruments Corporation
- *NI-VXI User Manual*, 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 Functional Body Specification*
- VPP-3.3, *Instrument Driver Interactive Developer Interface Specification*
- VPP-3.4, *Instrument Driver Programmatic Developer Interface 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*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.

1

Introduction

This chapter discusses how to use this manual, lists what you need to get started, and contains a brief description of the VISA Library. The National Instruments implementation of VISA is known as *NI-VISA*.

How to Use This Manual

This manual provides 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 the next section, *What You Need to Get Started*. The Read Me First document included with your kit explains how to install and set up your system.

Once you have set up your system, you can use Chapter 2 to guide yourself through some simple examples. Chapters 3 through 8 contain more in-depth information about the different elements that make up the VISA system.

What You Need to Get Started

- ❑ Appropriate hardware, in the form of a National Instruments GPIB, GPIB-VXI, MXI/VXI or serial interface board. For serial support, the computer's standard serial ports are sufficient.
- ❑ NI-488.2 and/or NI-VXI installed on your system. For serial support, the system's serial drivers are sufficient.
- ❑ 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:\VXIpnp\Win95\bin\HPGPVX32.dll`

Introduction to VISA

The main objective of the VXI*plug&play* Systems Alliance is to increase ease of use for end users of VXI technology through open, multivendor VXI 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 beyond the scope of the VXIbus specifications.

As a step toward industry-wide software compatibility, the alliance developed one specification for I/O software—the Virtual Instrument System Architecture, or VISA. The VISA specification defines a next-generation I/O software standard not only for VXI, but also for GPIB and serial interfaces. With the VISA standard endorsed by over 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. Before VISA, there were many different commercial implementations of I/O software for VXI, GPIB, and serial interfaces; however, none of these I/O software products were standardized or interoperable.

When the VISA standard was initially endorsed, commercial VISA products were not yet available. To quickly realize the benefits of VXI*plug&play*, the alliance developed the VISA Transition Library (VTL) specification. The VTL reflected the alliance's strategy to deliver multivendor software interoperability, while at the same time moving the entire industry towards a common, robust VISA foundation for the future. Software written to VTL, such as instrument drivers and executable soft front panels, will also run on present and future VISA implementations without modification.

All VXI*plug&play* products are classified within a framework. The concept of a framework was developed by the VXI*plug&play* Systems Alliance 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 test and measurement 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, ADEs, and programming languages into distinct frameworks and defined in-depth specifications to guarantee interoperability of the components within each framework.

This manual 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 Table 1-1. For information on programming VISA from LabVIEW, refer to the VISA documentation included with your LabVIEW software.

Table 1-1. NI-VISA Support

Operating System	Programming Language/ Environment	Framework
Windows 3.*x*	LabWindows/CVI, ANSI C, Visual Basic	WIN
	LabVIEW	GWIN
Windows 95	LabWindows/CVI, ANSI C, Visual Basic	WIN95
	LabVIEW	GWIN95
Windows NT	LabWindows/CVI, ANSI C, Visual Basic	WINNT
	LabVIEW	GWINNT
Solaris 1.*x*	LabWindows/CVI, ANSI C	SUN
Solaris 2.*x*	LabVIEW	GSUN
HP-UX 9	ANSI C, LabWindows/CVI*	HPUX
HP-UX 10	LabVIEW	GHPUX
Mac 68K	ANSI C	**
Mac PPC	LabVIEW	**
VxWorks	ANSI C	**

* Although the LabWindows/CVI development environment is not available on HP-UX, the run-time libraries are. Therefore, a LabWindows/CVI application developed on another framework can be ported to HP-UX without modification.

** This framework is not defined by the VXI*plug&play* Systems Alliance, but is still supported by NI-VISA.

You may find that programming with NI-VISA is not significantly different from programming with the I/O software products that are currently available. However, the programming concepts, model, and paradigm that NI-VISA uses create a solid foundation for taking advantage of VISA's more powerful features in the future.

2 Introductory Programming Examples

This chapter introduces some examples of common communication with instruments. To help you become comfortable with VISA, the examples avoid VISA terminology. Chapter 3, *VISA Overview*, looks at these examples again but using VISA terminology and focusing more on how they explain the VISA model.

Note ***The examples in this chapter show C source code. You can find the same examples in Visual Basic syntax in Appendix A, Visual Basic Examples.***

Example of Message-Based Communication

Serial, GPIB, and VXI systems all have a definition of message-based communication. In GPIB and serial, 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.

Example 2-1 shows the basic steps in any VISA program.

Example 2-1

```
#include "visa.h"

#define MAX_CNT 200

int main(void)
{
       ViStatus    status;             /* For checking errors         */
       ViSession   defaultRM, instr;   /* Communication channels      */
       ViUInt32    retCount;           /* Return count from string I/O*/
       ViChar      buffer[MAX_CNT];    /* Buffer for string I/O       */


       /* Begin by initializing the system                            */
       status = viOpenDefaultRM(&defaultRM);
       if (status < VI_SUCCESS) {
          /* Error Initializing VISA...exiting                        */
          return -1;
       }

       /* Open communication with GPIB Device at Primary Addr 1       */
       /* NOTE: For simplicity, we will not show error checking       */
       status = viOpen(defaultRM, "GPIB0::1::INSTR", VI_NULL, VI_NULL,
                       &instr);

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

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

       /* Your code should process the data                           */

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

Example 2-1 Discussion

We can break down Example 2-1 into the following steps.

1. Begin by initializing the VISA system. For this task you use `viOpenDefaultRM()`, which opens a communication channel with VISA itself. This channel has a purpose similar to a telephone line. The function call is analogous to picking up the phone and dialing the operator. From this point on, the phone line connects you to the driver. Any communication on the line is between you and the driver only. Chapter 3, *VISA Overview*, 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()` 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 which GPIB board you want. 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 Chapter 4, *Initializing Your VISA Application*.

   The two `VI_NULL`s 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 open channels to devices. You do not use this handle again until you are ready to end the program.

3. At this point you need to 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. 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.

   For more information on these functions, see Chapter 5, *Message-Based Communication*. Also refer to the NI-VISA online help or the *NI-VISA Programmer Reference Manual*.

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 returned by `viOpenDefaultRM()`.

Example of Register-Based Communication

Note ***You can skip over this section if you are exclusively using GPIB or serial 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 *Low-Level Access Operations* section in Chapter 6, *Register-Based Communication*.

Note ***Examples 2-2 through 2-4 use*** **`bold text`** ***to distinguish lines of code that are different from the other examples in this chapter.***

Example 2-2

```
#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 < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                       */
         return -1;
      }

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

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

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

Example 2-2 Discussion

The general structure of this example is very similar to that of Example 2-1. 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 Example 2-1 are replaced with register functions.

The address string is still the same format as the address string in Example 2-1, 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 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.

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 (an asynchronous event, or simply an *event*). Examples of this are the service request (SRQ), interrupts, and signals. 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 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, look at the *Callbacks* section in Chapter 7, *VISA Events*.

Queuing

When using a software queue, the 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. Example 2-3 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 2-3

```
#include "visa.h"

int main(void)
{
      ViStatus    status;             /* For checking errors          */
      ViSession   defaultRM, instr;   /* Communication channels       */
      ViEventType eventType;          /* To identify event            */
      ViEvent     eventData;          /* To hold event info           */
      ViUInt16    statID;             /* Interrupt Status ID          */

      /* Begin by initializing the system                              */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                          */
         return -1;
      }

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

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

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

      status = viWaitOnEvent(instr, VI_EVENT_VXI_SIGP, 5000, &eventType,
                             &eventData);
      if (status < VI_SUCCESS) {
         /* No interrupts received after 5000 ms timeout               */
         viClose(defaultRM);
         return -1;
      }

      /* Obtain the information about the event and then destroy the   */
      /* event. In this case, we want the status ID from the interrupt.*/
      status = viGetAttribute(eventData, VI_ATTR_SIGP_STATUS_ID, &statID);
      status = viClose(eventData);

      /* Your code should read data from the instrument and process it.*/
      /* Stop listening to events                                    */
      status = viDisableEvent(instr, VI_EVENT_VXI_SIGP, VI_QUEUE);

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

Example 2-3 Discussion

As you can see, this programming code presents some new functions you need to use. The first two functions you will notice are `viEnableEvent()` and `viDisableEvent()`. These functions tell the VISA driver which events to listen for—in this case the `VI_EVENT_VXI_SIGP`, or VXI Signal Processor events. These events cover both VXI interrupts and VXI signals. In addition, these functions tell the driver how to handle the 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. This shows that the VISA driver performs all event handling on a per-communication-channel basis.

When 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. However, after you have set the device up to interrupt when it is ready, 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 until the event occurs. Therefore, after the `viWaitOnEvent()` call is finished, either it has timed out (5 s in the above example) or it has caught the interrupt. After some error checking to determine which case is true and 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 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 the VISA specification requires that the driver close any channels opened off a channel to the driver (`defaultRM`) when the driver channel is closed. As a result, when you need to shut down a program quickly, such as in the case of an error, you can simply close the channel to the driver and VISA handles the rest of the details for you. However, VISA does not clean up anything not associated with VISA, such as memory you have allocated. You are still responsible for those items.

Example of Locking

Occasionally you may need to prevent other applications from using the same resource that you are using. VISA has a service called *locking* that you can use to gain exclusive access to a resource. VISA also has another locking option in which you can have multiple sessions share a lock. Because lock sharing is an advanced topic that may involve inter-process communication, see the *Lock Sharing* section in Chapter 8, *VISA Locks*, for more information. Example 2-4 uses the simpler form, the exclusive lock, to prevent other VISA applications from modifying the state of the specified serial port.

Example 2-4

```
#include "visa.h"

#define MAX_CNT 200

int main(void)
{
      ViStatus    status;             /* For checking errors         */
      ViSession   defaultRM, instr;   /* Communication channels      */
      ViUInt32    retCount;           /* Return count from string I/O*/
      ViChar      buffer[MAX_CNT];    /* Buffer for string I/O       */


      /* Begin by initializing the system                            */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                        */
         return -1;
      }

      /* Open communication with Serial Port 1                       */
      /* NOTE: For simplicity, we will not show error checking       */
      status = viOpen(defaultRM, "ASRL1::INSTR", VI_NULL, VI_NULL, &instr);
```

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

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

    /* Set serial port settings as needed                           */
    /* Defaults = 9600 Baud, no parity, 8 data bits, 1 stop bit     */
    status = viSetAttribute(instr, VI_ATTR_ASRL_BAUD, 2400);
    status = viSetAttribute(instr, VI_ATTR_ASRL_DATA_BITS, 7);

    /* 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;
}
```

Example 2-4 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 Chapter 8, *VISA Locks*. 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 a VXI*plug&play* instrument driver. Failure to do so may cause other applications to block or terminate with a failure.

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.

3

VISA Overview

This chapter contains an overview of the VISA Library.

Introduction

The history of instrumentation reached a milestone with the ability to communicate with an instrument from a remote computer. Before this time, you had to perform data collection and analysis manually through the controls on the instrument's front panel. 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. As time went on, the substantial programming task was alleviated by application development environments such as LabVIEW and LabWindows/CVI. These applications increased productivity, but instrumentation system developers were still faced with the details of programming the instrument or the device interface bus.

The VISA Library significantly reduces the time and effort involved in programming different interfaces. Instead of using a different Application Programmer's Interface (API) devoted to each interface bus, you can use the VISA API whether your system uses a GPIB, VXI, GPIB-VXI, PXI, or serial controller.

As an example, consider the case of the GPIB-VXI controller. You can use this device to communicate with VXI devices, but through a GPIB cable. In other words, you use a GPIB interface with GPIB software to send commands to VXI devices. There is no way for you to ignore the interface through which you must communicate. If you want to access the registers on the VXI device, you must use GPIB string communication to ask the GPIB-VXI to perform this action. Indeed, the specification of the GPIB-VXI (VXI-5) does not even standardize the commands necessary to do this task.

Objectives of VISA

The main objective of the VXI*plug&play* Systems Alliance is to increase ease of use for end users of VXI technology through open, multivendor VXI systems. Instrument programmers need a software architecture that exports the capabilities of the *devices*, not the interface bus. In addition, they need to be consistent across the devices and interface buses. Realizing these goals results in a simpler model to understand and reduces the number of functions the user needs to learn.

Using the example of the GPIB-VXI, a software driver that satisfies these goals should be capable of sending and receiving messages (string communication) to and from message-based devices. In addition, the communication functions should be the same, regardless of the interface through which these messages are sent. Any functionality that the device exports—such as message or register communication—should be accessible regardless of the capabilities of the interface bus. Moreover, you should be able to access this functionality through the *same functions* regardless of the interface bus you are using.

With the vast number of choices in instrumentation and software now available, most users do not want to be limited to a specific vendor for their systems. Instead, they prefer the freedom to select the best instruments and software available from multiple vendors and have it all work together with minimal effort. The IEEE 488.1 and IEEE 488.2 standards for GPIB and the IEEE 1155 standard for VXI ensured that the hardware would be interoperable, but such standards did not apply to the software. Therefore, the ideal new driver architecture should be a standard adopted by as many of the major vendors as possible to ensure that any code written for your instrument is portable across vendors as well as operating systems.

Finally, most instruments export a set of commands to which they will respond. Because the instrument needs to be flexible, 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, programmers are faced with a lot of overhead. Rather than making a simple request to *get the data*, one must issue a series of commands to *do task A*, *do task B*, and so on, prior to making the actual request to get the data.

National Instruments began to ease this burden with the development of *instrument drivers*, which encapsulate these primitive commands inside functions to perform the common tasks so users get up and running much faster. The major drawback has been that it is difficult to keep up with the number of new devices that appear in the marketplace. So, another objective for this ideal driver would be for it to be an accepted standard for creating instrument drivers. Then the vendors of the instruments could create the instrument drivers themselves and be assured that they can cover most of the systems on the market.

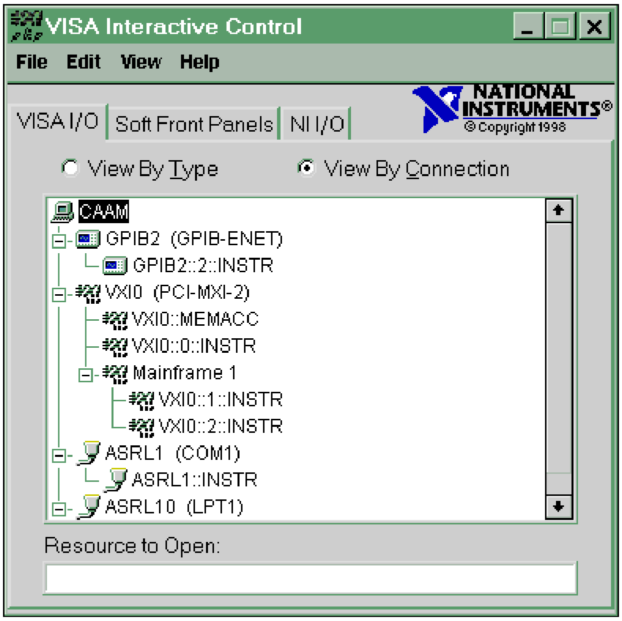
The VXI*plug&play* Systems Alliance formed to create this software architecture. The name of the driver is *VISA*, for Virtual Instrument Software Architecture. With VISA, you can benefit from the interface-independence features and the newly defined standard for instrument drivers. Future versions of VISA will support more advanced features, such as finer control of instruments and distribution across networks.

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 of VISA's functionality interactively, in an easy-to-use graphical environment. It is a convenient starting point for program development and learning about VISA.

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.

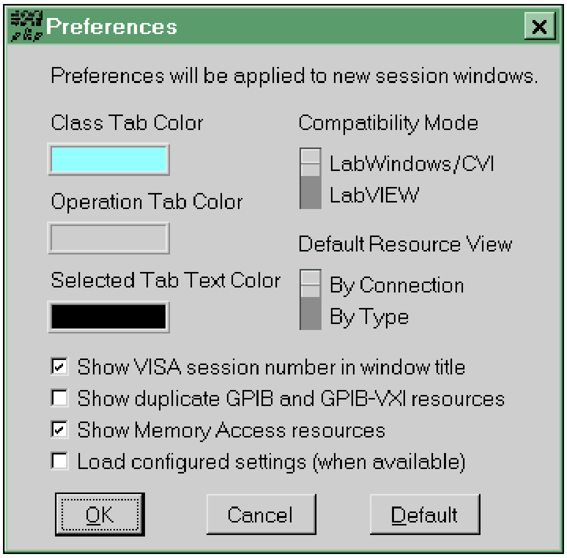
The following figure shows the VISAIC opening window.



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 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.

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. 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

Preferences will be applied to new session windows.

Class Tab Color

Operation Tab Color

Selected Tab Text Color

Compatibility Mode
LabWindows/CVI
LabVIEW

Default Resource View
By Connection
By Type

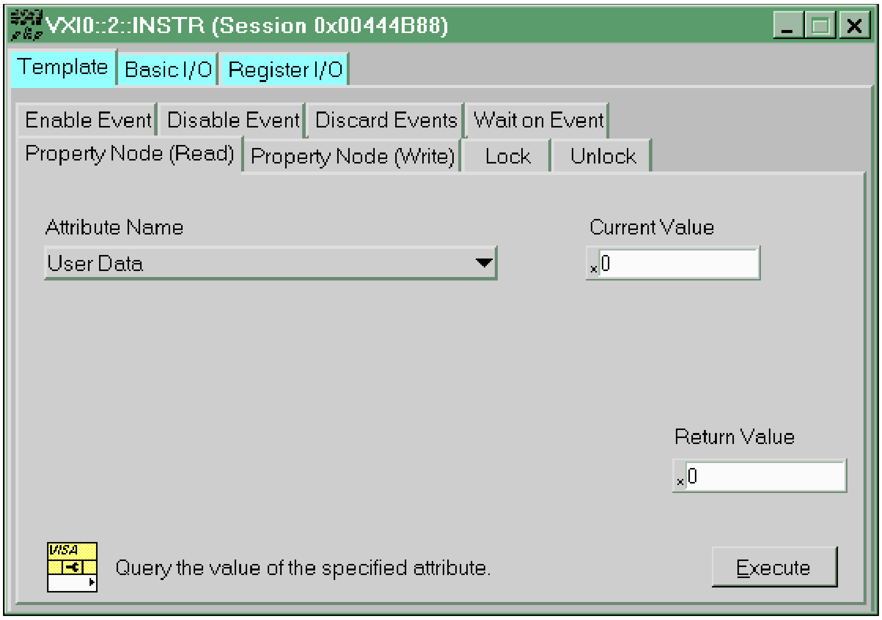
[✓] Show VISA session number in window title
[ ] Show duplicate GPIB and GPIB-VXI resources
[✓] Show Memory Access resources
[ ] Load configured settings (when available)

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 95/NT 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.



Attribute Name: User Data

Current Value: 0

Return Value: 0

Query the value of the specified attribute.

Three main tabs appear in the window. The initial tab is the **Template** tab, which contains all of the operations dealing with events, properties, and locks. Notice that there is a separate tab for each of these operations under the main tab. The other main tabs are **Basic I/O** and **Register I/O**. The **Basic I/O** tab contains the operations for message-based instruments, while the **Register I/O** tab contains the operations for register-based instruments. The **Register I/O** tab does not appear for either GPIB or Serial instruments.

Programming with VISA

Chapter 2, *Introductory Programming Examples*, introduced some examples of how to write code for the VISA driver. However, the chapter 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. This section looks at these examples again, but this time from the perspective of the underlying architecture.

Beginning Terminology

Let us begin by defining some 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. Therefore, when creating the building blocks for VISA, it is important to focus on these basic device capabilities. 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 what the termination character is, if any, and 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 termination character, *attributes*.

An important resource under VISA 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.

Another resource type is the Memory Access, or *MEMACC* Resource. The MEMACC Resource allows interface-level accesses, such as that used by NI-VXI, but is still independent of the actual interface type.

Returning to Example 2-1 in Chapter 2, *Introductory Programming Examples*, look at the point where the program has opened a communication channel with a message-based device. Remember that because of interface independence, it does not matter whether the device is GPIB or VXI. 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 no one ever receives 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);
```

You have just set an attribute (`VI_ATTR_TMO_VALUE`) of the resource. From now on, your communication to this resource through this communication channel (`instr`) will have a timeout of 5 s.

The fact that you are dealing with an OO-based driver is somewhat irrelevant at this point. As you become more experienced with VISA, you will see more of the benefits of this 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:

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

Again, this is a familiar approach to programming. You use a write operation to send a string to a device. For now, it is sufficient for you to understand that you can use a single operation—`viWrite()`—to send a message to a device, regardless of the interface to which it is connected.

Continuing, you read back the string with a read operation:

```
status = viRead(instr, buffer, 200, &retCount);
```

See Chapter 5, *Message-Based Communication*, for more information.

Communication Channels

The examples from Chapter 2, *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 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 talking to the resource and is therefore a *global attribute*. However, different programs may have different timeout requirements and so the timeout value (`VI_ATTR_TMO_VALUE`) for communication is a *local attribute*.

Again, look at Example 2-1. 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 both 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 a *resource* known as 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 previous section briefly mentioned the Resource Manager Resource. 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 Example 2-1, notice that the first line of code is a function call to get a session to the Default Resource Manager:

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

Note `viOpenDefaultRM()` ***is a* function *call, not an* operation *call.***

An operation is a property of a resource that you call by way of a session. 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 property of any resource—not even the Resource Manager Resource. It is provided by the VISA driver itself.

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 Chapter 4, *Initializing Your VISA Application*, for more information about `viFindRsrc()`.

Register Communication

Now that you know more about communicating with message-based devices, you can move on to register communication. The only types of devices VISA supports that can export register accesses are VXI and PXI devices. However, VISA has defined these resources to be expandable to other types of devices in the future.

Refer to Example 2-2 in Chapter 2, *Introductory Programming Examples*. You open communication to the Resource Manager and the resource in the same manner as in Example 2-1, but this time you specify a VXI device. This example uses what are known as the *High-Level Access* methods to read and write registers. For example, if you want to read a 16-bit register—say the ID register of the device—use the following call:

```
status = viIn16(instr, VI_A16_SPACE, 0x0, &deviceID);
```

Notice that the offset requested is 0. This is the offset of the ID register for a VXI device, but it is not the absolute address of the register in A16 space. This is because `instr` is a session to the instrument, not to the VXI memory space; therefore, all offsets are from the base address of the instrument. For example, if this same device also shared its memory in A24 space at 0x200000, you could write to the first memory location of this shared memory as follows:

```
status = viOut16(instr, VI_A24_SPACE, 0x0, 0x1234);
```

Thus, the offset when using an INSTR resource is 0x0, not 0x200000. VISA also supports sessions to the interface bus itself via the MEMACC Resource. Refer to Chapter 6, *Register-Based Communication*, for more information about this resource.

These methods are known as the *High-Level* Access methods to distinguish them from the *Low-Level* Access methods. The High-Level Access methods encapsulate all the necessary code to perform a read or a write of a register, including mapping any necessary access windows and handling errors, such as a bus error. In contrast, the Low-Level Access methods do not. Instead, you map the windows yourself and VISA does not monitor for errors. Refer to Chapter 6, *Register-Based Communication*, for more information about accessing register-based devices with both the High-Level Access and the Low-Level Access methods.

Example of Interface Independence

Now that you are more familiar with the architecture of the VISA driver, look at the GPIB-VXI interface board to see if VISA gives you independence from the interface connecting the instruments.

The GPIB-VXI device translates GPIB bus communication to VXIbus communication and vice versa. Its main purpose is to let GPIB users add VXI devices to their systems inexpensively. Using GPIB driver software, you can communicate with VXI devices using *messages*, the same way you program stand-alone GPIB instruments.

But how do you perform register accesses to the VXI devices? Up to this point, you were required to send messages to the GPIB-VXI itself and ask it to perform the register access. It would then return the result of the I/O in another string. 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 function calls:

```
dev = ibdev(boardID, PrimAddr, SecAddr, TMO, EOT, EOS);
status = ibwrt(dev, "A24 #h200000, #h1234", cnt);
```

If you are using NI-488.2 routines, the same call is:

```
Send(boardID, Address, "A24 #h200000, #h1234", DABend);
```

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 the same for all three cases. Therefore the code for writing to the A24 register through a GPIB-VXI is now precisely the same as given previously in the *Register Communication* section:

```
status = viOut16(instr, VI_A24_SPACE, 0x0, 0x1234);
```

The fact that GPIB messages are necessary is no longer important; you can let the driver take care of those details. Program your instrument based on its capabilities.

4

Initializing Your VISA Application

This chapter describes the steps required to prepare your application for communication with your device.

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.

Note ***The examples in this chapter show C source code. You can find the same examples in Visual Basic syntax in Appendix A, Visual Basic Examples.***

Example 4-1

```
#include "visa.h"

int main(void)
{
      ViStatus   status;
      ViSession  defaultRM, instr;

      /* Open Default RM                                              */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                         */
         return -1;
      }

      /* Access other resources                                       */
      status = viOpen(defaultRM, "GPIB::1::INSTR", VI_NULL, VI_NULL,
                      &instr);

      /* Use device and eventually close it.                          */
      viClose(instr);
      viClose(defaultRM);
      return 0;
}
```

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. The following table shows the format for this string. Square brackets indicate optional string segments.

Interface	Syntax
VXI	`VXI[`*`board`*`]::`*`VXI logical address`*`[::INSTR]`
GPIB-VXI	`GPIB-VXI[`*`board`*`]::`*`VXI logical address`*`[::INSTR]`
GPIB	`GPIB[`*`board`*`]::`*`primary address`*`[::`*`secondary address`*`][::INSTR]`
ASRL	`ASRL[`*`board`*`][::INSTR]`
VXI	`VXI[`*`board`*`]::MEMACC`
GPIB-VXI	`GPIB-VXI[`*`board`*`]::MEMACC`

Use the VXI keyword for VXI instruments via either embedded or MXIbus controllers. Use the GPIB-VXI keyword for a GPIB-VXI controller. Use the GPIB keyword to establish communication with a GPIB device. Use the ASRL keyword to establish communication with an asynchronous serial (such as RS-232) device.

Refer to Chapter 9, *NI-VISA Platform-Specific and Portability Issues*, 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 default values for optional string segments are as follows.

Optional String Segment	Default Value
`board`	0
`secondary address`	none

The following table shows examples of address strings.

Address String	Description
`VXI0::1::INSTR`	A VXI device at logical address 1 in VXI interface VXI0
`GPIB-VXI::9::INSTR`	A VXI device at logical address 9 in a GPIB-VXI controlled VXI system
`GPIB::1::0::INSTR`	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0
`ASRL1::INSTR`	A serial device attached to interface ASRL1
`VXI::MEMACC`	Board-level register access to the VXI interface
`GPIB-VXI1::MEMACC`	Board-level register access to GPIB-VXI interface number 1

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.

Notice that while this sample function returns a session, it does not return the reference to the resource manager session opened within the same function. If you use this style of initialization routine, you can get the reference to the resource manager session later by querying the attribute `VI_ATTR_RM_SESSION` before closing the INSTR session. You can then close the resource manager session with `viClose()`.

Example 4-2

```
#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;
      ViChar     desc[VI_FIND_BUFLEN];
      ViUInt32   numInstrs;
      ViUInt16   iManf, iModel;

      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error initializing VISA ... exiting                         */
         return status;
      }
      /* Find all VXI instruments in the system                         */
      status = viFindRsrc(defaultRM, "?*VXI[0-9]*::?*INSTR", &fList,
                          &numInstrs, desc);
```

```
    if (status < VI_SUCCESS) {
       /* Error finding resources ... exiting                        */
       viClose(defaultRM);
       return status;
    }

    /* Open a session to each and determine if it matches            */
    while (numInstrs--) {
       status = viOpen(defaultRM, desc, VI_NULL, VI_NULL, &instr);
       if (status < VI_SUCCESS) {
          viFindNext(fList, desc);
          continue;
       }
       status = viGetAttribute(instr, VI_ATTR_MANF_ID, &iManf);
       if ((status < VI_SUCCESS) || (iManf != MANF_ID)) {
          viClose(instr);
          viFindNext(fList, desc);
          continue;
       }
       status = viGetAttribute(instr, VI_ATTR_MODEL_CODE, &iModel);
       if ((status < VI_SUCCESS) || (iModel != MODEL_CODE)) {
          viClose(instr);
          viFindNext(fList, desc);
          continue;
       }

       /* We have a match, return the session without closing it     */
       *instrSesn = instr;
       viClose(fList);
       /* Do not close defaultRM, as that would close instr too      */
       return VI_SUCCESS;
    }

    /* No match was found, return an error                           */
    viClose(fList);
    viClose(defaultRM);
    return VI_ERROR_RSRC_NFOUND;
}
```

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`.

Finding VISA Resources Using Regular Expressions

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

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

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

Special Characters and Operators	Meaning
`exp|exp`	Matches either the preceding or following expression. The or operator `|` matches the entire expression that precedes or follows it and not just the character that precedes or follows it. For example, `VXI|GPIB` means `(VXI)|(GPIB)`, not `VX(I|G)PIB`.
`(exp)`	Grouping characters or expressions.

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

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

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
`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`.

Regular Expression	Sample Matches
`?*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.

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 Example 4-2, 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 4-3

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

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

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

      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error initializing VISA ... exiting                        */
         return status;
      }

      /* Find the first matching VXI instrument                        */
      sprintf(regExToUse,
      "?*VXI[0-9]*::?*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;
}
```

Configuring a Session

After the Resource Manager opens a session, communication with the device can usually begin using the default session settings. However, in some cases such as ASRL (serial) resources, you need to set some other parameters such as baud rate, parity, and flow control before proper communication can begin. GPIB and VXI sessions may have still other configuration parameters to set, such as timeouts and end-of-transmission modes, although in general the default settings should suffice.

Accessing Attributes

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

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

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

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

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

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 device. If you attempted to use it, the status parameter would return with the code `VI_ERROR_NSUP_ATTR`.

Refer to the online help or to the *NI-VISA Programmer Reference Manual* 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.
- `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. The following two VXI attributes are important for high-level accesses:

- `VI_ATTR_DEST_BYTE_ORDER` specifies whether to write words in big endian or little endian byte order. Defaults to `VI_BIG_ENDIAN`.
- `VI_ATTR_SRC_BYTE_ORDER` specifies whether to read words in big endian or little endian byte order. Defaults to `VI_BIG_ENDIAN`.

Because ASRL devices have much more variation in the communication channel, be sure to set the following parameters correctly:

- `VI_ATTR_ASRL_BAUD` sets the baud rate. Defaults to 9600.
- `VI_ATTR_ASRL_DATA_BITS` sets the number of data bits. Defaults to 8.
- `VI_ATTR_ASRL_PARITY` sets the parity. Defaults to `VI_ASRL_PAR_NONE`.
- `VI_ATTR_ASRL_STOP_BITS` sets the number of stop bits. Defaults to `VI_ASRL_STOP_ONE` (10).
- `VI_ATTR_ASRL_FLOW_CNTRL` sets the method for limiting overflow on transfers between the devices. Defaults to `VI_ASRL_FLOW_NONE` (no method of flow control).

Check the *Serial Port Support* section in Chapter 9, *NI-VISA Platform-Specific and Portability Issues*, to verify you are establishing connection to the correct port. Refer to the online help or to the *NI-VISA Programmer Reference Manual* for a complete range of values for the attributes. Some other useful ASRL 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`.
- `VI_ATTR_ASRL_END_OUT` defines the method of terminating writes. Defaults to `VI_ASRL_END_NONE`. This means that the setting of `VI_ATTR_SEND_EN` is irrelevant.

5 Message-Based Communication

This chapter shows how to use the VISA library in message-based communication.

Introduction

Whether you are using RS-232, GPIB, or VXI, message-based communication is a standard protocol for controlling and receiving data from instruments. Because most message-based devices have similar capabilities, it is natural that the driver interface should be consistent. Under VISA, controlling message-based devices is the same regardless of whether those devices are serial, GPIB, or VXI instruments.

VISA message-based communication includes the Basic I/O Services and the Formatted I/O Services from within the VISA Instrument Control Resource (INSTR). All sessions to a VISA Instrument Control Resource (INSTR) opened using `viOpen()` have full message-based communication capabilities. Of course, if the device is a register-based VXI device, the message-based operations return an error code (`VI_ERROR_NSUP_OPER`) to indicate that this *device* does not support the operations, although the *session* still provides access to them. This chapter 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 serial instruments, the I/O protocol must be set to*** `VI_ASRL_488` ***for the clear, trigger, and status services to be enabled.***

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

Synchronous Read/Write Services

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

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

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

The various ways that a string can be sent is the next issue to consider in message-based communication. For example, the actual mechanism for sending a byte differs drastically between GPIB and VXI; however, both have similar mechanisms to indicate when the last byte has been transferred. Under both systems, a device can specify an actual character, such as linefeed, to indicate that no more data will be sent. This is known as the End Of String (EOS) character and is common in older GPIB devices. The obvious drawback to this mechanism is that you must send an extra character to terminate the communication, and you cannot use this character in your messages. However, both GPIB and VXI can specify that the current byte is the last byte. GPIB uses the EOI line on the bus, and VXI uses the END bit in the Word Serial command that encapsulates the byte.

You need to determine how to inform the driver which mechanism to use. As was discussed in Chapter 3, *VISA Overview*, VISA uses a technique known as *attributes* to hold this information. For example, to tell the driver to use the EOI line or END bit, you set the `VI_ATTR_SEND_END_EN` attribute to true.

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

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

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

Refer to the NI-VISA online help or the *NI-VISA Programmer Reference Manual* 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 the *NI-VISA Programmer Reference Manual* or to the NI-VISA online help. For a more detailed example using asynchronous I/O, see Example 7-1 in Chapter 7, *VISA Events*.

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 a serial device, the controller sends the string `"*CLS\n"`. The I/O protocol must be set to `VI_ASRL_488` for this service to be available to serial devices.

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. Future versions of VISA will give you full access to interface triggering, but at this time all functionality is defined on a per-device basis.

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 Chapter 2, *Introductory Programming Examples*, and Chapter 7, *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                                                   */
```

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 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 the section *Controlling the Serial I/O Buffers* later in this chapter 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 `"F`*`x`*`"` command for function *X*. This could be `"F1"` for volt measurement, `"F2"` for ohm measurement, and so on. With formatted I/O, you can select the type of measurement and use only a single operation to send the string. Consider the following code segment.

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

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

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

or

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

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

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

I/O Buffer Operations

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

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

As you can see, the formatted I/O approach is the simplest way to get the job done. Because of the variety of modifiers you can use in the format string, this section does not go into any more detail on these operations. Please refer either to the NI-VISA online help or to Chapter 5, *Operations,* in the *NI-VISA Programmer Reference Manual* 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. Please 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 the description of the `viFlush()` operation in the NI-VISA online help or in the *NI-VISA Programmer Reference Manual* for more information.

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.

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);
```

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 output or input serial buffers by invoking the `viSetBuf()` operation with the `VI_ASRL_OUT_BUF` or `VI_ASRL_IN_BUF` flag, respectively. When the buffer size is non-zero, I/O to serial devices is not automatically flushed. You can force the output serial buffer to be flushed by invoking the `viFlush()` operation with `VI_ASRL_OUT_BUF`. Alternatively, you can call `viFlush()` with `VI_ASRL_OUT_BUF_DISCARD` to empty the output serial buffer without sending any remaining data to the device. You can also call `viFlush()` with either `VI_ASRL_IN_BUF` or `VI_ASRL_IN_BUF_DISCARD` to empty the input serial 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 serial input or output 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 serial output buffer. Similarly, `VI_WRITE_BUF_DISCARD` empties the contents of both the formatted I/O and low-level serial output buffers.

Example VISA Message-Based Application

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

Note ***This example shows C source code. You can find the same example in Visual Basic syntax in Appendix A,*** *Visual Basic Examples****.***

Example 5-1

```
#include "visa.h"

int main(void)
{
      ViSession   defaultRM, instr;
      ViUInt32    retCount;
      ViChar      idnResult[72];
      ViChar      resultBuffer[256];
      ViStatus    status;

      /* Open Default Resource Manager                                */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                         */
         return -1;
      }

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

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

6 Register-Based Communication

This chapter shows how to use the VISA library in register-based communication.

☞ **Note** ***You can skip this chapter if you are using GPIB or serial controllers exclusively. Register-based programming applies only to VXI or 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 the *Shared Memory Operations* section later in this chapter.

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 devices, VISA allows you to open a MEMACC (memory access) session. A session to a MEMACC Resource allows an application to access the entire memory range for a specified address space. The MEMACC Resource supports the same high-level and low-level operations as the INSTR Resource. The only difference is that all register addresses are absolute addresses in VXIbus address space.

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

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

	High-Level Access	High-Level Block	Low-Level Access

Read	`viIn8()`
	`viIn16()`
	`viIn32()`
		`viMoveIn8()`
		`viMoveIn16()`
		`viMoveIn32()`
			`viPeek8()`
			`viPeek16()`
			`viPeek32()`

Write	`viOut8()`
	`viOut16()`
	`viOut32()`
		`viMoveOut8()`
		`viMoveOut16()`
		`viMoveOut32()`
			`viPoke8()`
			`viPoke16()`
			`viPoke32()`

☞ **Note** ***The remainder of this chapter uses `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()`***.***

The following sections show the benefits of each style so you can make an informed choice of which is more appropriate for your programming requirements.

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 `viIn`*`XX`*`()` or `viOut`*`XX`*`()` 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.
- 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 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, please see Example 2-2 in Chapter 2, *Introductory Programming Examples*.

High-Level Block Operations

The high-level block operations `viMoveInXX()` and `viMoveOut`*`XX`*`()` 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 `viPeek`*`XX`*`()` or `viPoke`*`XX`*`()` operation invocations, instead of a single call to `viMoveIn`*`XX`*`()` or `viMoveOut`*`XX`*`()`.

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.
- The beginning offset of the registers relative to the device for the specified address space.

Note ***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 `viMoveIn`*`XX`*`()`) and `VI_ATTR_DEST_INCREMENT` (for `viMoveOut`*`XX`*`()`). 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 performs 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 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.

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.

To understand how mapping works, you must first remember that memory and address space are two different things. For example, most 32-bit CPUs have 4 GB of *address space*, but have *memory* measured in megabytes. This means that the CPU can put out over 2^{32} possible addresses onto the local bus, but only a small portion of that corresponds to memory. In most cases, the memory chips in the computer will respond to these addresses. However, because there is less memory in the computer than address space, National Instruments can add hardware that responds to other addresses. This hardware can then modify the address, according to the *mapping* that it has, to a VXI address and perform the access on the VXIbus automatically. The result is that the computer acts as if it is performing a local access, but in reality the access has been mapped out of the computer and to the VXIbus.

For example, consider an Intel 80x86-based computer running Windows. The addresses from 0xD0000 to 0xDFFFF (64 KB of addresses) do not correspond to any memory. You could add an AT-MXI board that listens for 0xD0000 to 0xDFFFF on the bus, and instruct it to map any addresses it finds in this range to the 64 KB of VXI A16 space. It does this by taking the *0xD* off the address so that it has a pure 64 KB address. For example, 0xDC000 would be mapped to 0xC000 in A16 space, which is the base address for a device at Logical Address 0. The same technique is used for other VXI address spaces as well. For example, if you wanted to access registers at 0x200000 in A24 space, you would tell the AT-MXI to strip off the *0xD* as before, but this time add 0x200000 to the resulting address and send it out 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 (which was 64 KB long in the previous example) 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 VXIbus.

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 VXI address space. The `viMapAddress()` operation first programs the hardware to map local CPU addresses to VXI 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 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.

Note ***To access the device registers through a 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 `viPeek`*`XX`*`()` and `viPoke`*`XX`*`()` 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—`viPeek`*`XX`*`()` and `viPoke`*`XX`*`()`—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. Example 6-1 shows how you can access other registers from `address`.

☞ **Note** ***The examples in this chapter show C source code. You can find the same examples in Visual Basic syntax in Appendix A, Visual Basic Examples.***

Example 6-1

```
#include "visa.h"

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

int main(void)
{
      ViStatus   status;              /* For checking errors        */
      ViSession  defaultRM, instr;    /* Communication channels     */
      ViAddr     address;             /* User pointer               */
      ViUInt16   value;               /* To store register value    */

      /* Begin by initializing the system                           */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                       */
         return -1;
      }

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

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

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

      status = viUnmapAddress(instr);

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

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

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 controller exports general-purpose memory to the A24/A32 space. In the second case, remote 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 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 Example 6-1. 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.

Note ***Example 6-2 uses* `bold text` *to distinguish lines of code that are different from those in Example 6-1.***

Example 6-2

```
#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) {
         /* 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,
                      &self);

      /* 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_NULL, &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;
}
```

7

VISA Events

This chapter describes the VISA event model and how to use it. The following sections discuss the various events VISA supports and the event handling paradigm.

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. The *Queuing* section in this chapter 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. The *Callbacks* section in this chapter 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

The following four events are currently defined for the Instrument Control Resource. These events do *not* apply to the Memory Access Resource.

- `VI_EVENT_SERVICE_REQ` (Service Request) is a notification of a service request from the device on a specific session.
- `VI_EVENT_VXI_SIGP` (VXI Signal Processor) is a notification of a VXIbus signal or VXIbus interrupt from the device. Notice that VISA supports the `VI_EVENT_VXI_SIGP` event only for VXI interfaces, so you can enable sessions only to VXI devices for this event.
- `VI_EVENT_VXI_VME_INTR` (VXI/VME Interrupt) is a notification of a VXIbus interrupt from the device. Notice that VISA supports the `VI_EVENT_VXI_VME_INTR` event only for VXI or VME interfaces, so you can enable sessions only to VXI or VME devices for this event.
- `VI_EVENT_TRIG` (VXI Trigger) is a notification of a VXIbus trigger. VXIbus interfaces support this event. Therefore, you can enable sessions only to VXI devices for this event.

VISA defines following two events for *both* the Instrument Control Resource and the Memory Access Resource.

- `VI_EVENT_IO_COMPLETION` (I/O Completion) is a notification that an asynchronous I/O operation has completed.

  The I/O Completion event applies to all asynchronous operations, which currently includes `viReadAsync()`, `viWriteAsync()`, and `viMoveAsync()`. You can use all three operations with the INSTR Resource but only `viMoveAsync()` with the MEMACC Resource.
- `VI_EVENT_EXCEPTION` (Exception) is a notification that an error condition has occurred during an operation invocation.

  The exception event supports only the callback model. Refer to the *Exception Handling* section at the end of this chapter for more information about this event type.

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.

All VISA events support the generic event attribute `VI_ATTR_EVENT_TYPE`. This attribute provides the type of the event—whether Service Request, VXI Signal Processor, VXI/VME Interrupt, VXIbus Trigger, or I/O Completion, or Exception.

In addition to this attribute, individual events may define attributes to hold additional event information. Currently, only the `VI_EVENT_SERVICE_REQ` event does *not* define additional attributes.

- `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_TRIG` defines `VI_ATTR_RECV_TRIG_ID`, which provides the trigger line on which the trigger was received.
- `VI_EVENT_IO_COMPLETION` defines, among other attributes, `VI_ATTR_STATUS` and `VI_ATTR_RET_COUNT`, which provide information about how the asynchronous I/O operation completed.
- `VI_EVENT_VXI_VME_INTR` defines `VI_ATTR_INTR_STATUS_ID` and `VI_ATTR_RECV_INTR_LEVEL`, which provide the interrupt status and interrupt level, respectively.
- `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 NI-VISA online help or to the *NI-VISA Programmer Reference Manual* 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 can use the `viEnableEvent()` operation to enable an event using either of the mechanisms. You can also enable events using a combination of both queuing and callback mechanisms by (bit-wise) ORing together the different mechanisms.

For example, to enable the `VI_EVENT_VXI_SIGP` event for queuing, use the following code:

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

However, to enable the same event for both queuing and callbacks, change the code as follows:

```
status = viEnableEvent(instr, VI_EVENT_VXI_SIGP,
                       VI_QUEUE | VI_HNDLR, VI_NULL);
```

Notice also that `viEnableEvent()` can add to the number of mechanisms in use during a session. For example, if you have enabled the application for queuing, it can make a subsequent call to `viEnableEvent()` specifying the callback mechanism. The end result is that *both* the queuing and callback mechanisms are enabled.

You cannot use the `viEnableEvent()` operation to decrease the number of mechanisms on which a session is enabled for sensing. Instead, you must use `viDisableEvent()` for that purpose. For example, if you have enabled a session for both `VI_QUEUE` and `VI_HNDLR`, a subsequent call to `viEnableEvent()` with the **mechanism** parameter set to `VI_QUEUE` does not change the mechanism to queuing only, but returns with the success code `VI_SUCCESS_EVENT_EN`, meaning that the specified event is enabled for at least one of the specified mechanisms. To disable the callback mechanism, call `viDisableEvent()` with its **mechanism** parameter set to `VI_HNDLR`. This action disables the callback mechanism but keeps the queuing method of notification enabled, as in the following example:

```
status = viDisableEvent(instr, VI_EVENT_VXI_SIGP, VI_HNDLR);
```

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.***

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. If you disable the queue, no further event occurrences are queued, 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 are of fixed length, but you can specify 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 on queue.

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

VISA does not currently let you dynamically configure queue lengths. That is, you can only modify the queue length 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 Example 2-3 in Chapter 2, *Introductory Programming Examples*, 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 the section *The userHandle Parameter* later in this chapter 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 earlier in the *Enabling and Disabling Events* section.
3. The driver invokes the handler on every occurrence of the specified event.
4. VISA provides the event context in the **context** parameter of `viEventHandler()`. The *event context* is like a data structure, and contains information about the specific occurrence of the event. Refer to the section *The Life of the Event Context* later in this chapter 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 earlier in the *Queuing* section of this chapter 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, say 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

Example 7-1 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.

Note ***This example shows C source code. You can find the same example in Visual Basic syntax in Appendix A, Visual Basic Examples.***

Example 7-1

```
#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 event, ViAddr userHandle)
{
      ViJobId     jobID;
      ViStatus    status;
      ViUInt16    stb;

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

int main(void)
{
      ViStatus    status;
      ViSession   defaultRM, gpibSesn;
      ViBuf       bufferHandle;
      ViUInt32    retCount;
      ViEventType etype;
      ViEvent     event;
      /* Begin by initializing the system                                */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error initializing VISA...exiting                            */
         return -1;
      }
      /* Open communication with GPIB device at primary address 2       */
      status = viOpen(defaultRM, "GPIB0::2::INSTR", VI_NULL, VI_NULL,
                     &gpibSesn);

      /* Allocate memory for buffer                                      */
```

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

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

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

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

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

    status = viWaitOnEvent(gpibSesn, VI_EVENT_IO_COMPLETION, 20000,
                           &etype, &event);
    if (status < VI_SUCCESS) {
       /* Waveform not received...exiting                             */
       free(bufferHandle);
       viClose(defaultRM);
       return -1;
    }
    /* Your code should process the waveform data                     */

    /* Close the event context                                        */
    viClose(event);
    /* 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;
}
```

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 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.

Event Context with the Callback Mechanism

In the case of callbacks, the event is passed to you in a function, so the 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 event, ViAddr uHandle)
{
      ViChar rsrcName[256], operName[256];
      ViStatus stat;
      ViSession rm;

      if (etype == VI_EVENT_EXCEPTION) {
         viGetAttribute(vi,VI_ATTR_RSRC_NAME,rsrcName);
         viGetAttribute(event,VI_ATTR_OPER_NAME,operName);
         viGetAttribute(event,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(event);
         viClose(vi);
         viClose(rm);
         exit(-1);   /* exit the application immediately */
      }
      /* code for other event types */
      return VI_SUCCESS;
}
```

If you wanted just to print out 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 wish to throw C++ exceptions from your handler.

8

VISA Locks

This chapter describes how to use locks in VISA.

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

Because the locking mechanism in VISA is session based, multiple threads sharing a session that has locked a VISA resource have the same privileges for accessing the resource. However, some applications might have separate sessions to a resource for these multiple threads, and might require that all the sessions in the application have the same privileges as the session that locked the resource. In other cases, there might be a need to share locks among sessions in different applications. Essentially, sessions that have a lock to a resource may share the lock with certain sessions, and exclude access from other sessions.

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

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

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

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

Acquiring an Exclusive Lock While Owning a Shared Lock

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

Nested Locks

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

Locking Sample Code

Example 8-1 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).

Note ***This example shows C source code. You can find the same example in Visual Basic syntax in Appendix A, Visual Basic Examples.***

Example 8-1

```
#include "visa.h"

#define MAX_COUNT 128

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

      /* Begin by initializing the system                             */
      status = viOpenDefaultRM(&defaultRM);
      if (status < VI_SUCCESS) {
         /* Error Initializing VISA...exiting                         */
         return -1;
      }
      /* Open communications with VXI Device at Logical Addr 16       */
      status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL,
                      &instrIN);
      status = viOpen(defaultRM, "VXI0::16::INSTR", VI_NULL, VI_NULL,
                      &instrOUT);

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

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

      status = viLock(instrIN, VI_SHARED_LOCK, 5000, VI_NULL, accKey);
      status = viLock(instrOUT, VI_SHARED_LOCK, VI_TMO_IMMEDIATE, accKey,
                      accKey);
      /* Set trigger channel for sessions                               */
      status = viSetAttribute(instrIN, VI_ATTR_TRIG_ID,VI_TRIG_TTL5);
      status = viSetAttribute(instrOUT,VI_ATTR_TRIG_ID,VI_TRIG_TTL4);

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

      /* Assert trigger to tell device to start sampling                */
      status = viAssertTrigger(instrOUT, VI_TRIG_PROT_DEFAULT);

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

      /* Close the event                                                */
      status = viClose(event);

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

      /* Your code should process the data                              */

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

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

9

NI-VISA Platform-Specific and Portability Issues

This chapter discusses 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 95/NT users**—On VXI and MXI systems, use T&M Explorer to run the VXI Resource Manager, configure your hardware, and assign VME and GPIB-VXI addresses. For GPIB systems, use the system Device Manager to configure your hardware. To control instruments through serial ports, you can use T&M Explorer 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 `vxiinit` and `resman`, and use `vxiedit` or `vxitedit` for configuration purposes. Similarly, for GPIB and GPIB-VXI systems, you still use the GPIB Control Panel applet or `ibconf` to configure your system. To control instruments through serial ports, you can do all necessary configuration at run-time by setting VISA attributes.

The *NI-VISA Programmer Reference Manual* contains detailed descriptions of the VISA attributes, events, and operations. Windows, Solaris, and HP-UX users can access this same information online through `NI-visa.hlp`, which you can find in the `NIvisa` directory.

Programming Considerations

This section contains information for you to consider when developing applications that use the NI-VISA I/O interface software.

Debugging Tool for Windows 95/NT

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-488.2 users may notice that NI Spy is similar to GPIB Spy.

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 so you can check them for errors at your convenience.

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 VXIbus access functions. The memory windows used to access the VXIbus are a limited resource. Call the `viMapAddress()` operation before attempting to perform low-level VXIbus 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 `viPeek`*`XX`*`()` and `viPoke`*`XX`*`()` on certain platforms. The C language macros automatically dereference the pointer whenever possible without calling the driver, which can substantially improve performance. The macros also handle any retry conditions on the new MXI-2 platforms. 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 `viPeek`*`XX`*`()` and `viPoke`*`XX`*`()` 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—available in C but not in LabVIEW or Visual Basic—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 `_far _pascal` for Windows 3.*x* (16-bit) and `_stdcall` for Windows 95 and Windows NT (32-bit). These are the standard Windows callback definitions. 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).

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 other than Macintosh, you can also make system calls. The way VISA invokes callbacks is platform dependent, as shown in Table 9-1.

Table 9-1. How VISA Invokes Callbacks

Platform	Callback Invocation Method
Windows 3.*x*	The application's stack and data segments are set up properly. The callback does not occur from within the driver interrupt service routine.
Windows 95 Windows NT	The callback is performed in a separate thread created by NI-VISA. The thread is signaled as soon as the event occurs.
Macintosh 68K Macintosh PPC	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.
Solaris 2.*x*	For VXI with the PCI-MXI-2, the callback is performed in a separate thread. For all other interfaces, the callback is performed via a UNIX signal.
VxWorks Solaris 1.*x* HP-UX 9 HP-UX 10	The callback is performed via a UNIX signal.

What this means is that on Windows 3.*x* (all interfaces) and Macintosh (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 these platforms, you must call any VISA operation or give up processor time. You can do this through any of the following methods (listed in order of portability):

1. Any VISA-defined operation
2. The LabWindows/CVI `ProcessSystemEvents()` function
3. The Windows `PeekMessage()` or `Yield()` functions

For example, the following code in a LabWindows/CVI application *does* allow callbacks to occur correctly.

```
while (!intr_recv)
   ProcessSystemEvents();   /* give up time */
```

Notice that NI-VISA on Windows 95, Windows NT, 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

NI-VISA supports all existing National Instruments VXI, GPIB, and serial hardware for the operating systems on which NI-VISA exists. For VXI, this includes MXI-1 and MXI-2 platforms, the GPIB-VXI, and the line of VXIpc embedded computers. For GPIB, this includes, but is not limited to, the PCI-GPIB, NB-GPIB, GPIB-SPARC series, the full line of AT-GPIB/TNT boards, and the GPIB-ENET box, which you can use to remotely control GPIB devices. With the GPIB-ENET, you can even remotely control VXI devices when using a GPIB-VXI controller.

Multiple GPIB-VXI Support

Windows 95/NT users can refer to the T&M Explorer utility to add multiple National Instruments GPIB-VXI controllers, or any other vendor’s GPIB-VXI controller, to your system. WIN16 and UNIX users must use the VISAconf utility to add the controllers.

Serial Port Support

The maximum number of serial ports that NI-VISA currently supports on any platform is 32. The default numbering of serial ports is system dependent, as shown in Table 9-2.

Table 9-2. How Serial Ports Are Numbered

Platform	Method
Windows 3.*x* Windows 95 Windows NT	`ASRL1`–`ASRL4` access `COM1`–`COM4`. `ASRL10`–`ASRL13` access `LPT1`–`LPT4`.
Macintosh 68K Macintosh PPC	`ASRL1` accesses the modem port. `ASRL2` accesses the printer port.
Solaris 2.*x*	`ASRL1`–`ASRL6` access `/dev/cua/a`–`/dev/cua/f`.
Solaris 1.*x*	`ASRL1`–`ASRL6` access `/dev/ttya`–`/dev/ttyf`.
HP-UX 9 HP-UX 10	`ASRL1` and `ASRL2` access serial ports 1 and 2 through `/dev/tty00` and `/dev/tty01`, respectively. Additional ports are numbered consecutively starting at `ASRL3`, which uses `/dev/tty02`.
VxWorks	NI-VISA for VxWorks does not currently support the serial interface.

VME Support

To access VME devices in your system, you must configure NI-VXI to see these devices. Windows 95/NT users can configure NI-VXI by using the **Add Device Wizard** in T&M Explorer. Users on other platforms must use the **Non-VXI Device Editor** in VXIedit or VXItedit. For each address space in which your device has memory, you must create a separate pseudo-device entry with a logical address between 256 and 511. For example, a VME device with memory in both A24 and A32 spaces requires two entries. You can also specify which interrupt levels the device uses. VXI and VME devices cannot share interrupt levels. You can then access the device from NI-VISA just as you would a VXI device, by specifying the address space and the offset from the base at which you have configured it. NI-VISA support for VME devices includes the register access operations (both high-level and low-level) and the block-move operations, as well as the ability to receive interrupts.

Windows 3.*x* Issues

This section contains information specific to Windows 3.*x* about the installation and use of NI-VISA.

Installation Overview

After the NI-VISA driver is installed, the Setup program normally makes some modifications to your initialization files `AUTOEXEC.BAT` and `WIN.INI`. If you choose *not* to let the installer make these changes automatically, the NI-VISA driver may not perform properly.

The necessary changes include adding the VXI*plug&play* binary directory (`C:\VXIPNP\WIN\BIN` by default) to the `PATH` environment variable in `AUTOEXEC.BAT`, and setting the `VPNPPATH` environment variable in both files to the root of the VXI*plug&play* directory tree (`C:\` by default).

Memory Model

The NI-VISA driver was compiled using the large memory model. However, Windows application programs that link with the VISA library can also use the medium, compact, or small memory models. Because of this ability to use different memory models for your application, not only can you take advantage of the efficiency inherent in small memory model programs, but you can also run multiple instances of the application.

Application Stack Size

The default stack size in Borland C++ is 5 KB, and in Microsoft Visual C++ it is 2 KB. In VISA, where the invocation of an operation may make other calls that in turn call a lower-level driver such as NI-VXI or NI-488.2, such a small stack may easily be exhausted, resulting in a stack overflow. For Windows 3.*x* (16-bit) VISA applications, set the stack size to a minimum of 8 KB using the `STACKSIZE` statement in the application's `.DEF` file. In LabWindows/CVI for Windows 3.*x*, the stack size is not normally a problem, as the default stack size is set to a more reasonable 16 KB.

A

Visual Basic Examples

This appendix shows the Visual Basic syntax of the ANSI C examples given earlier in this manual. The examples use the same numbering sequence for easy reference.

These examples use the VISA data types where applicable. This feature is available only on Windows 95/NT. 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.

♦ **Windows 3.*x* users**—Use the native Visual Basic types as described in the NI-VISA online help or *NI-VISA Programmer Reference Manual* in the *Data Types* section.

Example 2-1

```
Private Sub vbMain()
    Const MAX_CNT = 200

    Dim stat     As ViStatus
    Dim dfltRM   As ViSession
    Dim sesn     As ViSession
    Dim retCount As Long
    Dim buffer   As String * MAX_CNT

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

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

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

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

    Rem Your code should process the data

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

Example 2-2

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

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

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

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

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

Example 2-3

```
Private Sub vbMain()
    Dim stat   As ViStatus
    Dim dfltRM As ViSession
    Dim sesn   As ViSession
    Dim eType  As ViEventType
    Dim eData  As ViEvent
    Dim statID As Integer

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

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

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

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

    stat = viWaitOnEvent(sesn, VI_EVENT_VXI_SIGP, 5000, eType, eData)
    If (stat < VI_SUCCESS) Then
        Rem No interrupts received after 5000 ms timeout
        stat = viClose (dfltRM)
        Exit Sub
    End If

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

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


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

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

Example 2-4

```
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
```

Example 4-1

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

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

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

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

Example 4-2

```
Rem Find the first matching device and return a session to it
Private Function AutoConnect(instrSesn As ViSession) As ViStatus
    Const MANF_ID    = &HFF6 '12-bit VXI manufacturer ID of a device
    Const MODEL_CODE = &H0FE '12-bit or 16-bit model code of a device

    Dim stat    As ViStatus
    Dim dfltRM  As ViSession
    Dim sesn    As ViSession
    Dim fList   As ViFindList
    Dim desc    As String * VI_FIND_BUFLEN
    Dim nList   As Long
    Dim iManf   As Integer
    Dim iModel  As Integer

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

    Rem Find all VXI instruments in the system
    stat = viFindRsrc(dfltRM, "?*VXI[0-9]*::?*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
```

Example 4-3

Example 4-3 uses functionality not available in Visual Basic. Refer to Example 4-2 for sample code using `viFindRsrc()`.

Example 5-1

```
Private Sub vbMain()
    Dim stat          As ViStatus
    Dim dfltRM        As ViSession
    Dim sesn          As ViSession
    Dim retCount      As Long
    Dim idnResult     As String * 72
    Dim resultBuffer  As String * 256

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

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

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

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

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

    Rem Clear the device
    stat = viClear(sesn)

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

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

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

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

    Rem Close sessions
    stat = viClose (sesn)
    stat = viClose (dfltRM)
End Sub
```

Example 6-1

```
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
```

Example 6-2

```
Private Sub vbMain()
    Dim stat   As ViStatus
    Dim dfltRM As ViSession
    Dim self   As ViSession
    Dim addr   As ViAddr
    Dim offs   As Long
    Dim mSpace As Integer
    Dim Value  As Integer

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

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

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

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

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

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

Example 7-1

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

```
Private Sub vbMain()
    Const MAX_CNT = 1024

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

    Rem Begin by initializing the system
    Rem NOTE: For simplicity, we will not show error checking
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then
        Rem Error initializing VISA...exiting
        Exit Sub
    End If

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

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

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

    Rem Tell the device to begin acquiring a waveform
    stat = viWrite(sesn, "E0x51; W1", 9, retCount)

    Rem The device asserts SRQ when the waveform is ready
     stat = viWaitOnEvent(sesn, VI_EVENT_SERVICE_REQ, 20000, etype, event)
    If (stat < VI_SUCCESS) Then
        Rem Waveform not received...exiting
        stat = viClose (dfltRM)
        Exit Sub
    End If
    stat = viReadSTB (sesn, stb)
    Rem Read the data
    stat = viRead(sesn, bufferHandle, MAX_CNT, retCount)

    Rem Your code should process the waveform data

    Rem Close the event context
    stat = viClose (event)

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

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

Example 8-1

```
Private Sub vbMain()
    Const MAX_COUNT = 128
    Dim stat     As ViStatus                'For checking errors
    Dim dfltRM   As ViSession               'Communication channels
    Dim sesnIN   As ViSession               'Communication channels
    Dim sesnOUT  As ViSession               'Communication channels
    Dim aKey     As String * VI_FIND_BUFLEN  'Access key for lock
    Dim buf      As String * MAX_COUNT      'To store device data
    Dim etype    As ViEventType             'To identify event
    Dim event    As ViEvent                 'To hold event info
    Dim retCount As Long                    'To hold byte count
    Rem Begin by initializing the system
    stat = viOpenDefaultRM(dfltRM)
    If (stat < VI_SUCCESS) Then
        Rem Error initializing VISA...exiting
        Exit Sub
    End If
    Rem Open communications with VXI Device at Logical Addr 16
    stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesnIN)
    stat = viOpen(dfltRM, "VXI0::16::INSTR", VI_NULL, VI_NULL, sesnOUT)
    Rem We open two sessions to the same device
    Rem One session is used to assert triggers on TTL channel 4
    Rem The second is used to receive triggers on TTL channel 5
    Rem Lock first session as shared, have VISA generate the key
    Rem Then lock the second session with the same access key
    stat = viLock(sesnIN, VI_SHARED_LOCK, 5000, "", aKey)
    stat = viLock(sesnOUT, VI_SHARED_LOCK, VI_TMO_IMMEDIATE, aKey, aKey)
    Rem Set trigger channel for sessions
    stat = viSetAttribute(sesnIN,  VI_ATTR_TRIG_ID, VI_TRIG_TTL5)
    stat = viSetAttribute(sesnOUT, VI_ATTR_TRIG_ID, VI_TRIG_TTL4)
    Rem Enable input session for trigger events
    stat = viEnableEvent(sesnIN, VI_EVENT_TRIG, VI_QUEUE, VI_NULL)
    Rem Assert trigger to tell device to start sampling
    stat = viAssertTrigger(sesnOUT, VI_TRIG_PROT_DEFAULT)
    Rem Device will respond with a trigger when data is ready
    stat = viWaitOnEvent(sesnIN, VI_EVENT_TRIG, 20000, etype, event)
    If (stat < VI_SUCCESS) Then
        stat = viClose (dfltRM)
        Exit Sub
    End If

    Rem Close the event
    stat = viClose(event)

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

    Rem Your code should process the data

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

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

B Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
  Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	**Telephone**	**Fax**
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___) _________________Phone (___) ______________________________

Computer brand_____________ Model ____________________Processor______________________

Operating system (include version number) ______________________________

Clock speed ______MHz RAM _____MB Display adapter ______________________

Mouse ___yes ___no Other adapters installed______________________________

Hard disk capacity _____MB Brand______________________________

Instruments used ____

____

National Instruments hardware product model______________ Revision _____________________

Configuration ____

National Instruments software product ____________________ Version _____________________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem: ______________________________

____

____

____

____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *NI-VISA™ User Manual*

Edition Date: June 1998

Part Number: 321074D-01

Please comment on the completeness, clarity, and organization of the manual.

____

____

____

____

____

____

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

E-Mail Address ____

Phone (___) ____________________________ Fax (___) ____________________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meanings	Value
p-	pico	10^{-12}
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}
G-	giga-	10^{9}
t-	tera-	10^{12}

A

address location	Refers to the location of a specific register.
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.
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.
attribute	A value within an object or resource that reflects a characteristic of its operational state.

B

b	Bit
B	Byte

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.

C

callback	Same as *handler*. A software routine that is invoked when an asynchronous event occurs. In VISA, callbacks can be installed on any session that processes events.
commander	A device that has the ability to control another device. This term can also 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.
controller	An entity that can control another device(s) or is in the process of performing an operation on another device.

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. Same as 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.

E

event	An asynchronous occurrence that is independent of the normal sequential execution of the process running in a system.

F

FIFO	First In-First Out; a method of data storage in which the first element stored is the first one retrieved.

H

handler	Same as *callback*. A software routine that is invoked when an asynchronous event occurs. In VISA, callbacks can be installed on any session that processes events.

I

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.

L

lock	A state that prohibits sessions other than the session(s) owning the lock from accessing a resource.

M

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.

O

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.

P

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.

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.
resource class	The definition for how to create a particular resource. In 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 or resource instance	In general, this term is synonymous with the connotation of the word *object* in object-oriented architectures. For VISA, *resource* more specifically refers to a particular implementation (or *instance* in object-oriented terms) of a Resource Class.

S

s	second
session	The same as *communication channel*. A communication path between a software element and a resource. Every communication channel in VISA is unique.
shared library or shared object	Same as *DLL*. A file containing a collection of functions that can be used by multiple applications. This term is usually used for libraries on UNIX platforms.
shared memory	A block of memory that is accessible to both a client and a server. The memory block operates as a buffer for communication. This is unique to register-based interfaces such as VXI.

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 byte	A byte of information returned from a remote device that 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.

T

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).

V

virtual instrument	A name given to the grouping of software modules (in this case, VISA resources with any associated or required hardware) to give the functionality of a traditional stand-alone 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 instrument control resources	This is the name given to the part of VISA that defines all of the device-specific resource classes. VISA Instrument Control resources encompass all defined device capabilities for direct, low-level instrument control.
VISA memory access resources	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 resource manager	This is the name given to the part of VISA that manages resources. This management includes support for finding resources and opening sessions to them.
VISA resource template	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.

Index