Agilent E2090-90100
VISA Libraries

Agilent VISA User's Guide for Windows

Agilent Technologies

Notices

Manual Part Number

E2090-90100

Edition

Sixth edition, April 2003

Printed in USA

Agilent Technologies, Inc.
815 W 14th Street
Loveland, CO 80537 USA

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A **CAUTION** notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a **CAUTION** notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Agilent VISA User's Guide for Windows

1
Introduction

This *Agilent Technologies VISA User's Guide* describes the Agilent Virtual Instrument Software Architecture (VISA) library and shows how to use it to develop instrument drivers and I/O applications on Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT 4.0.

NOTE

Before you can use VISA, you must install and configure VISA on your computer. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for installation on Windows systems.

This guide shows programming techniques using C/C++ and Visual Basic. Since VISA and SICL are different libraries, using VISA functions and SICL functions in the same I/O application is not supported. Unless otherwise indicated, Windows NT refers to Windows NT 4.0.

This chapter includes:

- What's in This Guide?
- VISA Overview

What's in This Guide?

This chapter provides an overview of VISA and shows how to contact Agilent Technologies. Subsequent chapters in this guide address the following topics:

- *Chapter 2 - Building a VISA Application in Windows* describes how to build a VISA application in a Windows environment. An example program is provided to help you get started programming with VISA.
- *Chapter 3 - Programming with VISA* describes the basics of VISA and lists some example programs. The chapter also includes information on creating sessions, using formatted I/O, events, etc.
- *Chapter 4 - Programming via GPIB and VXI* provides guidelines for using VISA to communicate over the GPIB, GPIB-VXI, and VXI interfaces to instruments.
- *Chapter 5 - Programming via LAN* provides guidelines for using VISA to communicate over a LAN (Local Area Network) to instruments.
- *Chapter 6 - Programming via USB* provides guidelines for using VISA to communicate over a USB (Universal Serial Bus) to instruments.
- *Glossary* includes a glossary of terms and their definitions.

VISA Overview

VISA is a part of the Agilent IO Libraries. The Agilent IO Libraries consists of two libraries: *Agilent Virtual Instrument Software Architecture (VISA)* and *Agilent Standard Instrument Control Library (SICL)*. This guide describes VISA for supported Windows environments.

For information on using SICL in Windows, see the *Agilent SICL User's Guide for Windows*. For information on the Agilent IO Libraries, see the *Agilent IO Libraries Installation and Configuration Guide*.

Using VISA and SICL

Agilent Virtual Instrument Software Architecture (VISA) is an IO library designed according to the VXI*plug&play* System Alliance that allows software developed from different vendors to run on the same system.

Use VISA if you want to use VXI*plug&play* instrument drivers in your applications, or if you want the I/O applications or instrument drivers that you develop to be compliant with VXI*plug&play* standards. If you are using new instruments or are developing new I/O applications or instrument drivers, we recommend you use Agilent VISA.

Agilent Standard Instrument Control Library (SICL) is an I/O library developed by Agilent that is portable across many I/O interfaces and systems. You can use Agilent SICL if you have been using SICL and want to remain compatible with software currently implemented in SICL.

VISA Support

This 32-bit version of VISA is supported on Windows 98SE, Windows Me, Windows 2000, Windows XP, and Windows NT. Support for the 16-bit version of VISA was removed in version H.01.00 of the Agilent IO Libraries. However, versions through G.02.02 support 16-bit VISA. C, C++, and Visual Basic are supported on all these Windows versions.

For Windows, VISA is supported on the GPIB, VXI, GPIB-VXI, Serial (RS-232), LAN, and USB interfaces. VISA for the VXI interface on Windows NT is shipped with the Agilent Embedded VXI Controller product only. LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

VISA Users

VISA has two specific types of users. The first type is the instrumentation end user who wants to use VXI*plug&play* instrument drivers in his or her applications. The second type of user is the instrument driver or I/O application developer who wants to be compliant with VXI*plug&play* standards.

Software development using VISA is intended for instrument I/O and C/C++ or Visual Basic programmers who are familiar with the Windows 98SE, Windows Me, Windows 2000, Windows XP, or Windows NT environment. To perform VISA installation and configuration on Windows 2000, XP, or NT, you must have system administration privileges on the applicable system.

VISA Documentation

This table shows associated documentation you can use when programming with Agilent VISA.

Table 1 Agilent VISA Documentation

Document	Description
Agilent IO Libraries Installation and Configuration Guide for Windows	Shows how to install, configure, and maintain the Agilent IO Libraries on Windows.
VISA Online Help	Information is provided in the form of Windows Help.
VISA Example Programs	Example programs are provided online to help you develop VISA applications.
VXIplug&play System Alliance VISA Library Specification 4.3	Specifications for VISA.

Table 1 Agilent VISA Documentation

IEEE Standard Codes, Formats, Protocols, and Common Commands	ANSI/IEEE Standard 488.2-1992.
VXIbus Consortium specifications (when using VISA over LAN)	*TCP/IP Instrument Protocol Specification* - VXI-11, Rev. 1.0 *TCP/IP-VXIbus Interface Specification* - VXI-11.1, Rev. 1.0 *TCP/IP-IEEE 488.1 Interface Specification* - VXI-11.2, Rev. 1.0 *TCP/IP-IEEE 488.2 Instrument Interface Specification* - VXI-11.3, Rev. 1.0

Contacting Agilent

- In the USA and Canada, you can reach Agilent Technologies at these telephone numbers:

  USA: 1-800-452-4844
  Canada: 1-877-894-4414

- Outside the USA and Canada, contact your country's Agilent support organization. A list of contact information for other countries is available on the Agilent web site:

  `http://www.agilent.com/find/assist`

2
Building a VISA Application in Windows

This chapter provides guidelines for building a VISA application in a Windows environment.

The chapter contains the following sections:

- Building a VISA Program (C/C++)
- Building a VISA Program (Visual Basic)
- Logging Error Messages

Building a VISA Program (C/C++)

This section provides guidelines for building VISA programs using C/C++ language, including:

- Compiling and Linking VISA Programs (C/C++)
- Example VISA Program (C/C++)

Compiling and Linking VISA Programs (C/C++)

This section provides a summary of important compiler-specific considerations for several C/C++ compiler products when developing Win32 applications.

Linking to VISA Libraries

Your application must link to one of the VISA import libraries as follows, assuming default installation directories.

- **VISA on Windows 98SE or Windows Me:**

  ```
  C:\Program Files\VISA\win95\lib\msc\visa32.lib
        (Microsoft compilers)
  ```

  ```
  C:\Program Files\VISA\win95\lib\bc\visa32.lib
        (Borland compilers)
  ```

- **VISA on Windows 2000, XP, or NT:**

  ```
  C:\Program Files\VISA\winnt\lib\msc\visa32.lib
        (Microsoft compilers)
  ```

  ```
  C:\Program Files\VISA\winnt\lib\bc\visa32.lib
        (Borland compilers)
  ```

Microsoft Visual C++ Version 6.0 Compilers

1. Select **Project|Update All Dependencies** from the menu.
2. Select **Project|Settings** from the menu and click the **C/C++** button.
3. Select **Code Generation** from the **Category** list box and select **Multi-Threaded using DLL** from the **Use Run-Time Libraries** list

box. (VISA requires these definitions for Win32.) Click **OK** to close the dialog boxes.

4 Select **Project | Settings** from the menu. Click the **Link** button and add *visa32.lib* to the **Object/Library Modules** list box. Optionally, you may add the library directly to your project file. Click **OK** to close the dialog boxes.

5 You may want to add the *include files* and *library files* search paths. They are set as follows:

   - Select **Tools | Options** from the menu.
   - Click the **Directories** button to set the include file path.
   - Select **Include Files** from the **Show Directories For** list box.
   - Click the **Add** button and type one of the following: `C:\Program Files\VISA\win95\include` *or* `C:\Program Files\VISA\winnt\include`.

6 Select **Library Files** from the **Show Directories For** list box.

7 Click the **Add** button and type one of the following: `C:\Program Files\VISA\win95\lib\msc` *or* `C:\Program Files\VISA\winnt\lib\msc`

Borland C++ Version 4.0 Compilers

You may want to add the *include files* and *library files* search paths. They are set under the **Options|Project** menu selection. Double-click **Directories** from the **Topics** list box and add one of the following:

```
C:\Program Files\VISA\win95\include
        C:\Program Files\VISA\win95\lib\bc
```

or

```
C:\Program Files\VISA\winnt\include
        C:\Program Files\VISA\winnt\lib\bc
```

Example VISA Program (C/C++)

This section lists an example program called **idn** that queries a GPIB instrument for its identification string. This example assumes a Win32 Console Application using Microsoft or Borland C/C++ compilers on Windows.

- For VISA on Windows 98SE and Windows Me, the **idn** example files are in \Program Files\VISA\win95\agvisa\samples.
- For VISA on Windows 2000, XP, or NT, the idn example files are in \Program Files\VISA\winnt\agvisa\samples.

Example C/C++ Program Source Code

The source file **idn.c** follows. An explanation of the various function calls in the example is provided directly after the program listing. If the program runs correctly, the following is an example of the output if connected to a 54601A oscilloscope.

HEWLETT-PACKARD,54601A,0,1.7

If the program does not run, see the **Event Viewer** for a list of run-time errors.

```
/*idn.c
  This example program queries a GPIB device for
  an identification string and prints the
  results. Note that you must change the address.
*/

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

void main () {

  ViSession defaultRM, vi;
  char buf [256] = {0};

  /* Open session to GPIB device at address 22 */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM,
```

```
"GPIB0::22::INSTR",VI_NULL,VI_NULL,
         &vi);

   /* Initialize device */
   viPrintf(vi, "*RST\n");
   /* Send an *IDN? string to the device */
   viPrintf(vi, "*IDN?\n");
   /* Read results */
   viScanf(vi, "%t", buf);

   /* Print results */
   printf("Instrument identification string:
   %s\n", buf);

   /* Close session */
   viClose(vi);
   viClose(defaultRM);}
```

Example C/C++ Program Contents

A summary of the VISA function calls used in the preceding example C/C++ program follows. For a more detailed explanation of VISA functionality, see Chapter 3, *"Programming with VISA*." See the *VISA Online Help* for more detailed information on these VISA function calls.

Table 2 Summary of VISA Function Calls Used in the C/C++ Example

Function(s)	Description
visa.h	This file is included at the beginning of the program to provide the function prototypes and constants defined by VISA.
ViSession	The **ViSession** is a VISA data type. Each object that will establish a communication channel must be defined as **ViSession**.
viOpenDefaultRM	You must first open a session with the default resource manager with the **viOpenDefaultRM** function. This function will initialize the default resource manager and return a pointer to that resource manager session.

Table 2 Summary of VISA Function Calls Used in the C/C++ Example

viOpen	This function establishes a communication channel with the device specified. A session identifier that can be used with other VISA functions is returned. This call must be made for each device you will be using.
viPrintf and viScanf	These are the VISA formatted I/O functions that are patterned after those used in the C programming language. The **viPrintf** call sends the IEEE 488.2 ***RST** command to the instrument and puts it in a known state. The **viPrintf** call is used again to query for the device identification (***IDN?**). The **viScanf** call is then used to read the results.
viClose	This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.

Building a VISA Program (Visual Basic)

This section provides guidelines for building a VISA program in the Visual Basic language, including:

- Visual Basic Programming Considerations
- Example VISA Program (Visual Basic)

Visual Basic Programming Considerations

Some considerations for programming in Visual Basic follow.

Required Module for a Visual Basic VISA Program

Before you can use VISA specific functions, your application must add the *visa32.bas* VISA Visual Basic module found in one of the following directories (assuming default installation directories). For Windows 2000/XP/NT, `C:\Program Files\VISA\winnt\include\`. For Windows 98SE/Me, `C:\Program Files\VISA\win95\include\`.

Installing the visa32.bas File

To install *visa32.bas*:

1. Select **Project | Add Module** from the menu.
2. Select the **Existing** tab.
3. Browse and select the *visa32.bas* file from the applicable directory.
4. Click the **Open** button.

VISA Limitations in Visual Basic

VISA functions return a status code that indicates success or failure of the function. The only indication of an error is the value of a returned status code. The VB Error variable is not set by any VISA function. Thus, you cannot use the 'ON ERROR' construct in VB or the value of the VB Error variable to catch VISA function errors.

VISA cannot callback to a VB function. Thus, you can only use the **VI_QUEUE** mechanism in **viEnableEvent**. There is no way to install a VISA event handler in VB.

VISA functions that take a variable number of parameters (**viPrintf**, **viScanf**, **viQueryf**) are not callable from VB. Use the corresponding **viVPrintf**, **viVScanf** and **viVQueryf** functions instead.

You cannot pass variables of type *Variant* to VISA functions. If you attempt this, the Visual Basic program will probably crash with a 'General Protection Fault' or an 'Access Violation.'

Format Conversion Commands

The functions **viVPrintf**, **viVscanf** and **viVqueryf** can be called from VB, but there are restrictions on the format conversions that can be used. Only one format conversion command can be specified in a format string (a format conversion command begins with the % character).

For example, the following is invalid:

```
status = viVPrintf(vi, "%lf%d" + Chr$(10),
...)
```

Instead, you must make one call for each format conversion command, as shown in the following example:

```
status = viVPVISA User’s Guide for
Windowsntf(vi, "%lf" + Chr$(10), dbl_value)
status = viVPrintf(vi, "%d" + Chr$(10),
int_value)
```

Numeric Arrays

When reading from or writing to a numeric array, you must specify the first element of a numeric array as the *params* parameter. This passes the address of the first array element to the function. For example, the following code declares an array of 50 floating point numbers and then calls **viVPrintf** to write from the array.

```
Dim flt_array(50) As Double
status = viVPrintf(id, "%,50f", dbl_array(0))
```

Strings

When reading in a string value with **`viVScanf`** or **`viVQueryf`**, you must pass a fixed length string as the *params* parameter. To declare a fixed length string, instead of using the normal variable length declaration:

```
Dim strVal as String
```

use the following declaration, where 40 is the fixed length.

```
Dim strVal as String * 40
```

Example VISA Program (Visual Basic)

This section lists an example program called **`idn`** that queries a GPIB instrument for its identification string. This example builds a Standard *.exe* application for WIN32 programs using the Visual Basic 6.0 programming language.

For VISA on Windows 98SE/Me, the **idn** example files are in `C:\Program Files\VISA\win95\agvisa\samples\vb\idn`.

For VISA on Windows 2000/XP/NT, the **idn** example files are in `C:\Program Files\VISA\winnt\agvisa\samples\vb\idn`.

Steps to Running the Program

The steps to building and running the **idn** example program follow.

1 Connect an instrument to a GPIB interface that is compatible with IEEE 488.2.

2 Start the Visual Basic 6.0 application.

NOTE

This example assumes you are building a new project (no *.vbp* file exists for project). If you do not want to build the project from scratch, from the menu select **File | Open Project...** and select and open the *idn.vbp* file. Then skip to Step 9.

3 Start a new Visual Basic Standard *.exe* project. VB 6.0 will open a new project, **Project1** with a blank Form, **Form1**.

4 From the menu, select **Project | Add Module**, select the **Existing** tab, and browse to the `idn` directory.

5 The **idn** example files are located in directory `vb\samples\idn`. Select the file *idn.bas* and click **Open**. Since the Main( ) subroutine is executed when the program is run without requiring user interaction with a Form, **Form1** may be deleted if desired. To do this, right-click **Form1** in the Project Explorer window and select **Remove Form1**.

6 VISA applications in Visual Basic require the VISA Visual Basic (VB) declaration file *visa32.bas* in your VB project. This file contains the VISA function definitions and constant declarations needed to make VISA calls from Visual Basic.

7 To add this module to your project in VB 6.0, from the menu select **Project | Add Module**, select the **Existing** tab, browse to the directory containing the VB Declaration file, select *visa32.bas*, and click **Open**.

8 The name and location of the VB declaration file depends on which operating system is used. Assuming the 'standard' VISA directory `C:\Program Files\Visa`, or the 'standard' VXI*pnp* directory `C:\VXIpnp`, the *visa32.bas* file can be found in one of these locations:

\winnt\include\visa32.bas (Windows 2000/XP/NT)
\win95\include\visa32.bas (Windows 98SE/Me)

9 At this point, the Visual Basic project can be run and debugged. You will need to change the VISA Interface Name and address in the code to match your device's configuration.

10 If you want to compile to an executable file, from the menu select **File | Make idn.exe...** and press **Open**. This will create *idn.exe* in the **idn** directory.

Example Program Source Code

An explanation of the various function calls in the example is provided after this program listing. If the program runs correctly, the following is an example of the output in a Message Box if connected to a 54601A oscilloscope.

HEWLETT-PACKARD,54601A,0,1.7

If the program does not run, see the **Event Viewer** for a list of run-time errors. The source file idn.bas follows.

```
Option Explicit
''''''''''''''''''''''''''''''''''''''''''''''''
'''''''''''''''''''
' idn.bas
' This example program queries a GPIB device for
' an identification string and prints the
' results. Note that you may have to change the
' VISA Interface Name and address for your
' device from "GPIB0" and "22",respectively.
''''''''''''''''''''''''''''''''''''''''''''''''
'''''''''''''''''''

Sub Main()
   Dim defrm As Long  'Session to Default
   Resource Manager
   Dim vi As Long     'Session to instrument
   Dim strRes As String * 200  'Fixed length
   string to hold results

   ' Open the default resource manager session
   Call viOpenDefaultRM(defrm)
```

```
   ' Open the session to the resource
   ' The "GPIB0" parameter is the VISA Interface
   ' name to a
   ' GPIB instrument as defined in
   '    Start | Programs | Agilent IO Libraries |
      IO Config
   ' Change this name to what you have defined
   ' your VISA Interface.
   ' "GPIB0::22::INSTR" is the address string
   ' for the device.
   ' this address will be the same as seen in:
   ' Start | Programs | Agilent IO Libraries |
      VISA
   ' Assistant after the VISA Interface Name is
     defined in IO Config)

   Call viOpen(defrm, "GPIB0::22::INSTR", 0, 0,
   vi)

   ' Initialize device
   Call viVPrintf(vi, "*RST" + Chr$(10), 0)

   ' Ask for the device's *IDN string.
   Call viVPrintf(vi, "*IDN?" + Chr$(10), 0)

   ' Read the results as a string.
   Call viVScanf(vi, "%t", strRes)

   ' Display the results
   MsgBox "Result is: " + strRes, vbOKOnly,
   "*IDN? Result"

   ' Close the vi session and the resource
    manager session
   Call viClose(vi)
   Call viClose(defrm)
End Sub
```

Example Program Contents

A summary of the VISA function calls used in the preceding example Visual Basic program follows. For a more detailed explanation of VISA functionality, see Chapter 3, *"Programming with VISA."* See the *VISA Online Help* for more detailed information on these VISA function calls.

Table 3 Summary of VISA Function Calls in Visual Basic Example

Function(s)	Description
viOpenDefaultRM	You must first open a session with the default resource manager with the **viOpenDefaultRM** function. This function will initialize the default resource manager and return a pointer (*defrm*) to that resource manager session.
viOpen	This function establishes a communication channel with the device specified. A session identifier (**vi**) that can be used with other VISA functions is returned. This call must be made for each device you will be using.
viVPrintf and viVScanf	These are the VISA formatted I/O functions. The **viVPrintf** call sends the IEEE 488.2 ***RST** command to the instrument (plus a linefeed character) and puts it in a known state. The **viVPrintf** call is used again to query for the device identification (***IDN?**). The **viVScanf** call is then used to read the results (strRes) that are displayed in a Message Box.
viClose	This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.

Logging Error Messages

When developing or debugging your VISA application, you may want to view internal VISA messages while your application is running. You can do this by using the **Message Viewer** utility (for Windows 98SE/Me), the **Event Viewer** utility (for Windows 2000/XP/NT), or the **Debug Window** (for Windows 98SE/Me/2000/XP/NT). There are three choices for VISA logging:

- **Off** (default) for best performance
- **Event Viewer/Message Viewer**
- **Debug Window**

Using the Event Viewer

For Windows 2000, XP, or NT, the **Event Viewer** utility provides a way to view internal VISA error messages during application execution. Some of these internal messages do not represent programming errors and are actually error messages from VISA which are being handled internally by VISA. The process for using the **Event Viewer** is:

- Enable VISA logging from the Agilent IO Libraries Control by clicking the blue **IO** icon on the taskbar and then clicking **Agilent VISA Options | VISA Logging | Event Viewer**.
- Run your VISA program.
- View VISA error messages by running the **Event Viewer**. From the Agilent IO Libraries Control, click **Run Event Viewer**. VISA error messages will appear in the application log of the **Event Viewer** utility.

Using the Message Viewer

For Windows 98SE or Windows Me, the **Message Viewer** utility provides a way to view internal VISA error messages during application execution. Some of these internal messages do not represent programming errors and are actually error messages from VISA which are being handled internally by VISA.

The **Message Viewer** utility must be run BEFORE you run your VISA application. However, the utility will receive messages while minimized. This utility also provides menu selections for saving the logged messages to a file and for clearing the message buffer.

The process for using the **Message Viewer** is:

- Enable VISA logging from the Agilent IO Libraries Control by clicking the blue **IO** icon on the taskbar, then clicking **Agilent VISA Options | VISA Logging | Message Viewer**.
- Start the **Message Viewer**. From the Agilent IO Libraries Control, click **Run Message Viewer**.
- Run your VISA program.
- View error messages in the **Message Viewer** window.

Using the Debug Window

When VISA logging is directed to the **Debug Window**, VISA writes logging messages using the Win32 API call *OutputDebugString()*. The most common use for this feature is when debugging your VISA program using an application such as Microsoft Visual Studio. In this case, VISA messages will appear in the Visual Studio output window. The process for using the **Debug Window** is:

1. Enable VISA logging from the Agilent IO Libraries Control by clicking the blue **IO** icon on the taskbar and then clicking **Agilent VISA Options | VISA Logging | Debug Window**.
2. Run your VISA program from Microsoft Visual Studio (or equivalent application).
3. View error messages in the Visual Studio (or equivalent) output window.

3
Programming with VISA

This chapter describes how to program with VISA. The basics of VISA are described, including formatted I/O, events and handlers, attributes, and locking. Example programs are also provided and can be found in the `Samples` subdirectory on Windows environments.

See *VISA Library Information* in the *VISA Online Help* for the specific location of the example programs on your operating system. For specific details on VISA functions, see the *VISA Online Help*.

This chapter contains the following sections:

- VISA Resources and Attributes
- Using Sessions
- Sending I/O Commands
- Using Events and Handlers
- Trapping Errors
- Using Locks

VISA Resources and Attributes

This section introduces VISA resources and attributes, including:

- VISA Resources
- VISA Attributes

VISA Resources

In VISA, a **resource** is defined as any device (such as a voltmeter) with which VISA can provide communication. VISA defines six **resource classes** that a complete VISA system, fully compliant with the *VXIplug&play Systems Alliance* specification, can implement. Each resource class includes:

- **Attributes** to determine the state of a resource or session or to set a resource or session to a specified state.
- **Events** for communication with applications.
- **Operations** (functions) that can be used for the resource class.

A summary description of each resource class supported by Agilent VISA follows. See *VISA Resource Classes* in the *VISA Online Help* for a description of the attributes, events, and operations for each resource class.

NOTE Although the Servant Device-Side (SERVANT) resource is defined by the VISA specification, the SERVANT resource is not supported by Agilent VISA. See *VISA Resource Classes* in the *VISA Online Help* for a description of the SERVANT resource.

Table 4 Descriptions of Resource Classes Supported by Agilent VISA

Resource Class	Interface Types	Resource Class Description
Instrument Control (INSTR)	Generic, GPIB, GPIB-VXI, Serial, TCPIP, USB, VXI	Device operations (reading, writing, triggering, etc.).

Table 4 Descriptions of Resource Classes Supported by Agilent VISA

GPIB Bus Interface (INTFC)	Generic, GPIB	Raw GPIB interface operations (reading, writing, triggering, etc.).
Memory Access (MEMACC)	Generic, GPIB-VXI, VXI	Address space of a memory-mapped bus such as the VXIbus.
VXI Mainframe Backplane (BACKPLANE)	Generic, GPIB-VXI, VXI (GPIB-VXI BACKPLANE not supported)	VXI-defined operations and properties of each backplane (or chassis) in a VXIbus system.
Servant Device-Side Resource (SERVANT)	GPIB, VXI, TCPIP (not supported)	Operations and properties of the capabilities of a device and a device's view of the system in which it exists.
TCPIP Socket (SOCKET)	Generic, TCPIP	Operations and properties of a raw network socket connection using TCPIP.

VISA Attributes

Attributes are associated with **resources** or **sessions**. You can use attributes to determine the state of a resource or session, or to set a resource or session to a specified state.

For example, you can use the **viGetAttribute** function to read the state of an attribute for a specified session, event context, or find list. There are read only (RO) and read/write (RW) attributes. Use the **viSetAttribute** function to modify the state of a read/write attribute for a specified session, event context, or find list.

The pointer passed to **viGetAttribute** must point to the exact type required for that attribute (**ViUInt16**, **ViInt32**, etc). For example, when reading an attribute state that returns a **ViUInt16**, you must declare a variable of that type and use it for the returned data. If **ViString** is returned, you must allocate an array and pass a pointer to that array for the returned data.

Example: Reading a VISA Attribute

This example reads the state of the VI_ATTR_TERMCHAR_EN attribute and changes it if it is not true.

```
ViBoolean state, newstate;
newstate=VI_TRUE;
viGetAttribute(vi, VI_ATTR_TERMCHAR_EN, &state);
if (state err !=VI_TRUE) viSetAttribute(vi,
   VI_ATTR_TERMCHAR_EN, newstate);
```

Using Sessions

This section shows how to use VISA sessions, including:

- Including the VISA Declarations File (C/C++)
- Adding the *visa32.bas* File (Visual Basic)
- Opening a Session to a Resource
- Addressing a Session
- Closing a Session
- Searching for Resources

Including the VISA Declarations File (C/C++)

For C and C++ programs, you must include the *visa.h* header file at the beginning of every file that contains VISA function calls:

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

This header file contains the VISA function prototypes and the definitions for all VISA constants and error codes. The *visa.h* header file also includes the *visatype.h* header file.

The *visatype.h* header file defines most of the VISA types. The VISA types are used throughout VISA to specify data types used in the functions. For example, the **viOpenDefaultRM** function requires a pointer to a parameter of type **ViSession**. If you find **ViSession** in the *visatype.h* header file, you will find that **ViSession** is eventually typed as an unsigned long. VISA types are also listed in *VISA System Information* in the *VISA Online Help*.

Adding the visa32.bas File (Visual Basic)

You must add the *visa32.bas* Basic Module file to your Visual Basic Project. The *visa32.bas* file contains the VISA function prototypes and definitions for all VISA constants and error codes.

Opening a Session

A **session** is a channel of communication. Sessions must first be opened on the default resource manager, and then for each resource you will be using.

- A **resource manager session** is used to initialize the VISA system. It is a parent session that knows about all the opened sessions. A resource manager session must be opened before any other session can be opened.
- A **resource session** is used to communicate with a resource on an interface. A session must be opened for each resource you will be using. When you use a session you can communicate without worrying about the type of interface to which it is connected. This insulation makes applications more robust and portable across interfaces.

Resource Manager Sessions

There are two parts to opening a communications session with a specific resource. First, you must open a session to the default resource manager with the **viOpenDefaultRM** function. The first call to this function initializes the default resource manager and returns a session to that resource manager session. You only need to open the default manager session once. However, subsequent calls to **viOpenDefaultRM** returns a unique session to the same default resource manager resource.

Resource Sessions

Next, open a session with a specific resource using the **viOpen** function. This function uses the session returned from **viOpenDefaultRM** and returns its own session to identify the resource session. The following shows the function syntax.

```
viOpenDefaultRM(sesn);
viOpen(sesn, rsrcName, accessMode, timeout,
   vi);
```

The session returned from **viOpenDefaultRM** must be used in the *sesn* parameter of the **viOpen** function. The **viOpen** function then uses that session and the resource address specified in the *rsrcName* parameter to open a resource session. The *vi* parameter in **viOpen** returns a session identifier that can be used with other VISA functions.

Your program may have several sessions open at the same time after creating multiple session identifiers by calling the **viOpen** function multiple times. The following table summarizes the parameters in the previous function calls.

Table 5 Parameters Used in Function Calls

Parameter	Description
sesn	A session returned from the **viOpenDefaultRM** function that identifies the resource manager session.
rsrcName	A unique symbolic name of the resource (resource address).
accessMode	Specifies the modes by which the resource is to be accessed. The value VI_EXCLUSIVE_LOCK is used to acquire an exclusive lock immediately upon opening a session. If a lock cannot be acquired, the session is closed and an error is returned. The VI_LOAD_CONFIG value is used to configure attributes specified by some external configuration utility. If this value is not used, the session uses the default values provided by this specification. Multiple access modes can be used simultaneously by specifying a "bit-wise OR" of the values. (Must use VI_NULL in VISA 1.0.).
timeout	If the *accessMode* parameter requires a lock, this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Otherwise, this parameter is ignored. (Must use VI_NULL in VISA 1.0.)
vi	This is a pointer to the session identifier for this particular resource session. This pointer will be used to identify this resource session when using other VISA functions.

Example: Opening a Resource Session

This example shows one way of opening resource sessions with a GPIB multimeter and a GPIB-VXI scanner. The example first opens a session with the default resource manager. The session returned from the resource manager and a resource address is then used to open a session with the GPIB device at address 22. That session will now be identified as *dmm* when using other VISA functions.

The session returned from the resource manager is then used again with another resource address to open a session with the GPIB-VXI device at primary address 9 and VXI logical address 24. That session will now be identified as *scanner* when using other VISA functions. See the following section, “Addressing a Session” for information on addressing particular devices.

```
ViSession defaultRM, dmm, scanner;
.
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::22::INSTR",VI_NULL,
   VI_NULL,&dmm);
viOpen(defaultRM, "GPIB-VXI0::24::INSTR",
   VI_NULL, VI_NULL,&scanner);
.
viClose(scanner);
viClose(dmm);
viClose(defaultRM);
```

Addressing a Session

As shown in the previous section, the *rsrcName* parameter in the **viOpen** function is used to identify a specific resource. This parameter consists of the VISA interface name and the resource address. The interface name is determined when you run the VISA configuration utility. This name is usually the interface type followed by a number.

The following table illustrates the format of the *rsrcName* for different interface types. *INSTR* is an optional parameter that indicates that you are communicating with a resource that is of type INSTR, meaning instrument. The keywords are:

- **ASRL -** establishes communication with asynchronous serial devices.

- **GPIB -** establishes communication with GPIB devices or interfaces.
- **GPIB-VXI -** used for GPIB-VXI controllers.
- **TCPIP -** establishes communication with LAN instruments.
- **VXI -** used for VXI instruments.
- **USB -** used for USB instruments.

Table 6 The Format of the *rsrcName* for Different Interface Types

Interface	Typical Syntax
ASRL	ASRL[*board*][::INSTR]
GPIB	GPIB[*board*]::*primary address*[::*secondary address*][::INSTR]
GPIB	GPIB[*board*]::INTFC
GPIB-VXI	GPIB-VXI[*board*]::*VXI logical address*[::INSTR]
GPIB-VXI	GPIB-VXI[*board*]::MEMACC
GPIB-VXI	GPIB-VXI[*board*][::*VXI logical address*]::BACKPLANE
TCPIP	TCPIP[*board*]::*host address*[::*LAN device name*]::INSTR
TCPIP	TCPIP[*board*]::*host address*::*port*::SOCKET
USB	USB[*board*]::*manufacturer ID*::*model code*::*serial number*[::*USB interface number*][::INSTR]
VXI	VXI[*board*]::*VXI logical address*[::INSTR]
VXI	VXI[*board*]::MEMACC
VXI	VXI[*board*][::*VXI logical address*]::BACKPLANE

The following table describes the parameters used above.

Table 7 Description of Parameters

Parameter	Description
board	This optional parameter is used if you have more than one interface of the same type. The default value for *board* is 0.

Table 7 Description of Parameters

host address	The IP address (in dotted decimal notation) or the name of the host computer/gateway.
LAN device name	The assigned name for a LAN device. The default is *inst()*.
manufacturer ID	Manufacturer's ID for a USB Test & Measurement class device
model code	Model code of a USB device.
port	The port number to use for a TCP/IP Socket connection.
primary address	This is the primary address of the GPIB device.
secondary address	This optional parameter is the secondary address of the GPIB device. If no *secondary address* is specified, none is assumed.
serial number	Serial number of a USB device.
USB interface number	Interface number of a USB device.
VXI logical address	This is the logical address of the VXI instrument.

Some examples of valid symbolic names follow.

Table 8 Examples of Valid Symbolic Names

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 located on port 1.
VXI::MEMACC	Board-level register access to the VXI interface.

Table 8 Examples of Valid Symbolic Names

GPIB-VXI1::MEMACC	Board-level register access to GPIB-VXI interface number 1.
GPIB2::INTFC	Interface or raw resource for GPIB interface 2.
VXI::1::BACKPLANE	Mainframe resource for chassis 1 on the default VXI system, which is interface 0.
GPIB-VXI2:: BACKPLANE	Mainframe resource for default chassis on GPIB-VXI interface 2.
GPIB1::SERVANT	Servant/device-side resource for GPIB interface 1.
VXI0::SERVANT	Servant/device-side resource for VXI interface 0.
TCPIP0::1.2.3.4::999::SOCKET	Raw TCPIP access to port 999 at the specified address.
TCPIP::devicename@company.com::INSTR	TCPIP device using VXI-11 located at the specified address. This uses the default LAN Device Name of *inst0*.
USB::0x1234::125::A22-5::INSTR	A USB Test & Measurement class device with manufacturer ID 0x1234, model code 125, and serial number A22-5. This uses the device's first available USBTMC interface, which is usually number 0.

Example: Opening a Session

This example shows one way to open a resource session with the GPIB device at primary address 23.

```
ViSession defaultRM, vi;
.
.
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::23::INSTR", VI_NULL,
   VI_NULL,&vi);
.
```

```
.
viClose(vi);
viClose(defaultRM);
```

Closing a Session

The **viClose** function must be used to close each session. You can close the specific resource session, which will free all data structures that had been allocated for the session. If you close the default resource manager session, all sessions opened using that resource manager session will be closed.

Since system resources are also used when searching for resources (**viFindRsrc**), the **viClose** function needs to be called to free up find lists. See the following section, “Searching for Resources” for more information on closing find lists.

Searching for Resources

When you open the default resource manager, you are opening a parent session that knows about all the other resources in the system. Since the resource manager session knows about all resources, it has the ability to search for specific resources and open sessions to these resources. You can, for example, search an interface for devices and open a session with one of the devices found.

Use the **viFindRsrc** function to search an interface for device resources. This function finds matches and returns the number of matches found and a handle to the resources found. If there are more matches, use the **viFindNext** function with the handle returned from **viFindRsrc** to get the next match:

```
viFindRsrc(sesn, expr, findList, retcnt,
   instrDesc);
.
.
viFindNext(findList, instrDesc);
.
.
viClose (findList);
```

The parameters are defined as follows.

Table 9 Definitions of Parameters

Parameter	Description
sesn	The resource manager session.
expr	The expression that identifies what to search (see Table 10).
findList	A handle that identifies this search. This handle will then be used as an input to the **viFindNext** function when finding the next match.
retcnt	A pointer to the number of matches found.
instrDesc	A pointer to a string identifying the location of the match. Note that you must allocate storage for this string.

The handle returned from **viFindRsrc** should be closed to free up all the system resources associated with the search. To close the find object, pass the *findList* to the **viClose** function.

Use the *expr* parameter of the **viFindRsrc** function to specify the interface to search. You can search for devices on the specified interface. Use the following table to determine what to use for your *expr* parameter.

NOTE

Because VISA interprets strings as regular expressions, the string *GPIB?*INSTR* applies to *both* GPIB and GPIB-VXI devices.

Table 10 Determining What to Use for the *expr* Parameter

Interface	*expr* Parameter
GPIB	GPIB[0-9]*::?*INSTR
VXI	VXI?*INSTR
GPIB-VXI	GPIB-VXI?*INSTR
GPIB and GPIB-VXI	GPIB?*INSTR
All VXI	?*VXI[0-9]*::?*INSTR

Table 10 Determining What to Use for the *expr* Parameter

ASRL	ASRL[0-9]*::?*INSTR
All	?*INSTR

Example: Searching the VXI Interface for Resources

This example searches the VXI interface for resources. The number of matches found is returned in *nmatches*, and *matches* points to the string that contains the matches found. The first call returns the first match found, the second call returns the second match found, etc. VI_FIND_BUFLEN is defined in the *visa.h* declarations file.

```
ViChar buffer [VI_FIND_BUFLEN];
ViRsrc matches=buffer;
ViUInt32 nmatches;
ViFindList list;
.
.
viFindRsrc(defaultRM, "VXI?*INSTR", &list,
   &nmatches, matches);
..
.
viFindNext(list, matches);
.
.
viClose(list);
```

Sending I/O Commands

This section provides guidelines for sending I/O commands, including:

- Types of I/O
- Using Formatted I/O
- Using Non-Formatted I/O

Types of I/O

Once you have established a communications session with a device, you can start communicating with that device using VISA's I/O routines. VISA provides both formatted and non-formatted I/O routines.

- **Formatted I/O** converts mixed types of data under the control of a format string. The data is buffered, thus optimizing interface traffic.
- **Non-formatted I/O** sends or receives raw data to or from a device. With non-formatted I/O, no format or conversion of the data is performed. Thus, if formatted data is required, it must be done by the user.

You can choose between VISA's formatted and non-formatted I/O routines. However, since the non-formatted I/O performs the low-level I/O, you should not mix formatted I/O and non-formatted I/O in the same session. See the following sections for descriptions and examples using formatted I/O and non-formatted I/O in VISA.

Using Formatted I/O

The VISA formatted I/O mechanism is similar to the C **stdio** mechanism. The VISA formatted I/O functions are **viPrintf**, **viQueryf**, and **viScanf**. There are also two non-buffered and non-formatted I/O functions that synchronously transfer data, called **viRead** and **viWrite**, and two that asynchronously transfer data, called **viReadAsync** and **viWriteAsync**.

These are raw I/O functions and do not intermix with the formatted I/O functions. See "Using Non-Formatted I/O" in this chapter for details. See the *VISA Online Help* for more information on how data is converted under the control of the format string.

Formatted I/O Functions

As noted, the VISA formatted I/O functions are **viPrintf**, **viQueryf**, and **viScanf**.

- The **viPrintf** functions format according to the format string and send data to a device. The **viPrintf** function sends separate *arg* parameters, while the **viVPrintf** function sends a list of parameters in *params*:

  ```
  viPrintf(vi, writeFmt[, arg1][, arg2][, ...]);
  viVPrintf(vi, writeFmt, params);
  ```

- The **viScanf** functions receive and convert data according to the format string. The **viScanf** function receives separate *arg* parameters, while the **viVScanf** function receives a list of parameters in *params*:

  ```
  viScanf(vi, readFmt[, arg1][, arg2][, ...]);
  viVScanf(vi, readFmt, params);
  ```

- The **viQueryf** functions format and send data to a device and then immediately receive and convert the response data. Hence, the **viQueryf** function is a combination of the **viPrintf** and **viScanf** functions. Similarly, the **viVQueryf** function is a combination of the **viVPrintf** and **viVScanf** functions. The **viQueryf** function sends and receives separate *arg* parameters, while the **viVQueryf** function sends and receives a list of parameters in *params*:

  ```
  viQueryf(vi, writeFmt, readFmt[, arg1]
     [, arg2][, ...]);
  viVQueryf(vi, writeFmt, readFmt, params);
  ```

Formatted I/O Conversion

The formatted I/O functions convert data under the control of the format string. The format string specifies how the argument is converted before it is input or output. The format specifier sequence consists of a **%** (percent) followed by an optional modifier(s), followed by a format code.

```
%[modifiers]format code
```

Zero or more modifiers may be used to change the meaning of the format code. Modifiers are only used when sending or receiving formatted I/O. To send formatted I/O, the asterisk (*) can be used to indicate that the number is taken from the next argument.

However, when the asterisk is used when receiving formatted I/O, it indicates that the assignment is suppressed and the parameter is discarded. Use the pound sign (**#**) when receiving formatted I/O to indicate that an extra argument is used. The following are supported modifiers. See the **viPrintf** function in the *VISA Online Help* for additional enhanced modifiers (**@1, @2, @3, @H, @Q**, or **@B**).

Field Width Field width is an optional integer that specifies how many characters are in the field. If the **viPrintf** or **viQueryf** (*writeFmt*) formatted data has fewer characters than specified in the field width, it will be padded on the left, or on the right if the **– flag** is present.

You can use an asterisk (*) in place of the integer in **viPrintf** or **viQueryf** (*writeFmt*) to indicate that the integer is taken from the next argument. For the **viScanf** or **viQueryf** (*readFmt*) functions, you can use a **#** sign to indicate that the next argument is a reference to the field width.

The field width modifier is only supported with **viPrintf** and **viQueryf** (*writeFmt*) format codes **d, f, s**, and **viScanf** and **viQueryf** (*readFmt*) format codes **c**, **s**, and **[]**. (See Table 11 for a description of format codes.)

Example: Using Field Width Modifier

The following example pads **numb** to six characters and sends it to the session specified by *vi*:

```
int numb = 61;
viPrintf(vi, "%6d\n", numb);
```

Inserts four spaces, for a total of 6 characters: **61**

.Precision Precision is an optional integer preceded by a period. This modifier is only used with the **viPrintf** and **viQueryf** (*writeFmt*) functions. The meaning of this argument is dependent on the conversion character used. You can use an asterisk (*) in place of the integer to indicate the integer is taken from the next argument.

Table 11 Descriptions of Format Codes

Format Code	Description
d	Indicates the minimum number of digits to appear is specified for the **@1**, **@H**, **@Q**, and **@B** flags, and the **i**, **o**, **u**, **x**, and **X** format codes.
f	Indicates the maximum number of digits after the decimal point is specified.
s	Indicates the maximum number of characters for the string is specified.
g	Indicates the maximum significant digits are specified.

Example: Using the Precision Modifier

This example converts **numb** so that there are only two digits to the right of the decimal point and sends it to the session specified by *vi*:

```
float numb = 26.9345;
viPrintf(vi, "%.2f\n", numb);
```

Sends : **26.93**

Argument Length Modifier The meaning of the optional argument length modifier **h, l, L, z,** or **Z** is dependent on the conversion character, as listed in the following table. Note that **z** and **Z** are not ANSI C standard modifiers.

Table 12 Argument Length Modifiers

Argument Length Modifier	Format Codes	Description
h	d,b,B	Corresponding argument is a short integer or a reference to a short integer for **d**. For **b** or **B**, the argument is the location of a block of data or a reference to a data array. (**B** is only used with **viPrintf** or **viQueryf** (*writeFmt*).)
l	d,f,b,B	Corresponding argument is a long integer or a reference to a long integer for **d**. For **f**, the argument is a double float or a reference to a double float. For **b** or **B**, the argument is the location of a block of data or a reference to a data array. (**B** is only used with **viPrintf** or **viQueryf** (*writeFmt*).)
L	f	Corresponding argument is a long double or a reference to a long double.
z	b,B	Corresponding argument is an array of floats or a reference to an array of floats. (**B** is only used with **viPrintf** or **viQueryf** (*writeFmt*).)
Z	b,B	Corresponding argument is an array of double floats or a reference to an array of double floats. (**B** is only used with **viPrintf** or **viQueryf** (*writeFmt*).)

, Array Size The comma operator is a format modifier that allows you to read or write a comma-separated list of numbers (only valid with **%d** and **%f** format codes). It is a comma followed by an integer. The integer indicates the number of elements in the array. The comma operator has the format of *,dd* where *dd* is the number of elements to read or write.

For **viPrintf** or **viQueryf** (*writeFmt*), you can use an asterisk (*) in place of the integer to indicate that the integer is taken from the next argument. For **viScanf** or **viQueryf** (*readFmt*), you can use a **#** sign to indicate that the next argument is a reference to the array size.

Example: Using Array Size Modifier

This example specifies a comma-separated list to be sent to the session specified by *vi*:

```
int list[5]={101,102,103,104,105};
viPrintf(vi, "%,5d\n", list);
```

Sends: **101,102,103,104,105**

Special Characters Special formatting character sequences will send special characters. The following describes the special characters and what will be sent.

The format string for **viPrintf** and **viQueryf** (*writeFmt*) puts a special meaning on the newline character (*\n*). The newline character in the format string flushes the output buffer to the device.

All characters in the output buffer will be written to the device with an **END** indicator included with the last byte (the newline character). This means you can control at what point you want the data written to the device. If no newline character is included in the format string, the characters converted are stored in the output buffer. It will require another call to **viPrintf**, **viQueryf** (*writeFmt*), or **viFlush** to have those characters written to the device.

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes. The * while using the **viScanf** functions acts as an assignment suppression character. The input is not assigned to any parameters and is discarded.

The grouping operator **()** in a regular expression has the highest precedence, the + and * operators in a regular expression have the next highest precedence after the grouping operator, and the *or* operator **|** in a regular expression has the lowest precedence. The following table provides detailed descriptions of special characters and operators. Some example expressions follow in Table 14.

Table 13 Descriptions of Special Characters and Operators

<table>
<tr><th>Special Characters and Operators</th><th>Description</th></tr>
<tr><td>?</td><td>Matches any one character.</td></tr>
<tr><td>\</td><td>Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (e.g.,’ '\?’), it matches the '?' character instead of any one character.</td></tr>
<tr><td>[list]</td><td>Matches any one character from the enclosed list. A hyphen can be used to match a range of characters.</td></tr>
<tr><td>[^list]</td><td>Matches any character not in the enclosed list. A hyphen can be used to match a range of characters.</td></tr>
<tr><td>*</td><td>Matches 0 or more occurrences of the preceding character or expression.</td></tr>
<tr><td>+</td><td>Matches 1 or more occurrences of the preceding character or expression.</td></tr>
<tr><td>exp|exp</td><td>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 VXI(I|G)PIB.</td></tr>
<tr><td>(exp)</td><td>Grouping characters or expressions.</td></tr>
<tr><td>“ “</td><td>Sends a blank space.</td></tr>
<tr><td>\n</td><td>Sends the ASCII line feed character. The END identifier will also be sent.</td></tr>
<tr><td>\r</td><td>Sends an ASCII carriage return character.</td></tr>
<tr><td>\t</td><td>Sends an ASCII TAB character.</td></tr>
<tr><td>\###</td><td>Sends ASCII character specified by octal value.</td></tr>
</table>

Table 13 Descriptions of Special Characters and Operators

\"	Sends the ASCII double quote character.
\\	Sends a backslash character.

Table 14 Examples of Expressions and Matches

Example 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[0-9]::?*INSTR	Matches **GPIB0::2::INSTR** and **GPIB1::1::1::INSTR** but not **GPIB12::8::INSTR**
GPIB[^0]::?*INSTR	Matches **GPIB1::1::1::INSTR** but not **GPIB0::2::INSTR** or **GPIB12::8::INSTR**
VXI?*INSTR	Matches **VXI0::1::INSTR** but not **GPIB-VXI0::1::INSTR**
GPIB-VXI?*INSTR	Matches **GPIB-VXI0::1::INSTR** but not **VXI0::1::INSTR**
?*VXI[0-9]*::?*INSTR	Matches **VXI0::1::INSTR** and **GPIB-VXI0::1::INSTR**
ASRL[0-9]*::?*INSTR	Matches **ASRL1::INSTR** but not **VXI0::5::INSTR**
ASRL1+::INSTR	Matches **ASRL1::INSTR** and **ASRL11::INSTR** but not **ASRL2::INSTR**
(GPIB|VXI)?*INSTR	Matches **GPIB1::5::INSTR** and **VXI0::3::INSTR** but not **ASRL2::INSTR**
(GPIB0|VXI0)::1::INSTR	Matches **GPIB0::1::INSTR** and **VXI0::1::INSTR**
?*INSTR	Matches all **INSTR** (device) resources
?*VXI[0-9]*::?*MEMACC	Matches **VXI0::MEMACC** and **GPIB-VXI1::MEMACC**
VXI0::?*	Matches **VXI0::1::INSTR**, **VXI0::2::INSTR**, and **VXI0::MEMACC**
?*	Matches all resources

Format Codes. This table summarizes the format codes for sending and receiving formatted I/O.

Table 15 Format Codes for Sending and Receiving Formatted I/O

Format Codes	Description
viPrintf/viVPrintf and viQueryf/viVqueryf (*writeFmt*)	
d, i	Corresponding argument is an integer.
f	Corresponding argument is a double.
c	Corresponding argument is a character.
s	Corresponding argument is a pointer to a null terminated string.
%	Sends an ASCII percent (%) character.
o, u, x, X	Corresponding argument is an unsigned integer.
e, E, g, G	Corresponding argument is a double.
n	Corresponding argument is a pointer to an integer.
b, B	Corresponding argument is the location of a block of data.
viPrintf/viVPrintf and viQueryf/viVqueryf (*readFmt*)	
d,i,n	Corresponding argument must be a pointer to an integer.
e,f,g	Corresponding argument must be a pointer to a float.
c	Corresponding argument is a pointer to a character sequence.
s,t,T	Corresponding argument is a pointer to a string.
o,u,x	Corresponding argument must be a pointer to an unsigned integer.
[	Corresponding argument must be a character pointer.
b	Corresponding argument is a pointer to a data array.

Example: Receiving Data From a Session

This example receives data from the session specified by the *vi* parameter and converts the data to a string.

```
char data[180];
viScanf(vi, "%t", data);
```

Formatted I/O Buffers

The VISA software maintains both a read and write buffer for formatted I/O operations. Occasionally, you may want to control the actions of these buffers. You can modify the size of the buffer using the **viSetBuf** function. See the *VISA Online Help* for more information on this function.

The write buffer is maintained by the **viPrintf** or **viQueryf** (*writeFmt*) functions. The buffer queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string. It may occasionally be flushed at other non-deterministic times, such as when the buffer fills.

When the write buffer flushes, it sends its contents to the device. If you set the VI_ATTR_WR_BUF_OPER_MODE attribute to VI_FLUSH_ON_ACCESS, the write buffer will also be flushed every time a **viPrintf** or **viQueryf** operation completes. See "VISA Attributes" in this chapter for information on setting VISA attributes.

The read buffer is maintained by the **viScanf** and **viQueryf** (*readFmt*) functions. It queues the data received from a device until it is needed by the format string. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to **viScanf** or **viQueryf** reads data directly from the device rather than data that was previously queued.

If you set the VI_ATTR_RD_BUF_OPER_MODE attribute to VI_FLUSH_ON_ACCESS, the read buffer will be flushed every time a viScanf or viQueryf operation completes. See "VISA Attributes" in this chapter for information on setting VISA attributes.

You can manually flush the read and write buffers using the **viFlush** function. Flushing the read buffer also includes reading all pending response data from a device. If the device is still sending data, the flush process will continue to read data from the device until it receives an **END** indicator from the device.

Example: Sending and Receiving Formatted I/O

This C program example shows sending and receiving formatted I/O. The example opens a session with a GPIB device and sends a comma operator to send a comma-separated list. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter for more information.

This example program is installed on your system in the `Samples` subdirectory on Windows environments. See *VISA Library Information* in the *VISA Online Help* for locations of example programs on your operating system.

```
/*formatio.c
This example program makes a multimeter
measurement with a comma-separated list passed
with formatted I/O and prints the results. You
may need to change the device address. */

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

void main () {

  ViSession defaultRM, vi;
  double res;
  double list [2] = {1,0.001};

  /* Open session to GPIB device at address 22*/
  viOpenDefaultRM(&efaultRM);
  viOpen(defaultRM, "GPIB0::22::INSTR",
     VI_NULL,VI_NULL, &vi);

  /* Initialize device */
  viPrintf(vi, "*RST\n");
```

```
/* Set up device and send a comma-separated
list */
viPrintf(vi, "CALC:DBM:REF 50\n");
viPrintf(vi, "MEAS:VOLT:AC? %,2f\n", list);

/* Read results */
viScanf(vi, "%lf", &res);

/* Print results */
printf("Measurement Results: %lf\n", res);

/* Close session */
viClose(vi);
viClose(defaultRM);
}
```

Using Non-Formatted I/O

There are two non-buffered, non-formatted I/O functions that synchronously transfer data called **viRead** and **viWrite**. Also, there are two non-formatted I/O functions that asynchronously transfer data called **viReadAsync** and **viWriteAsync**. These are raw I/O functions and do not intermix with the formatted I/O functions.

Non-Formatted I/O Functions

The non-formatted I/O functions follow. For more information, see the **viRead**, **viWrite**, **viReadAsync**, **viWriteAsync**, and **viTerminate** functions in the *VISA Online Help*.

viRead. The **viRead** function synchronously reads raw data from the session specified by the *vi* parameter and stores the results in the location where *buf* is pointing. Only one synchronous read operation can occur at any one time.

```
viRead(vi, buf, count, retCount);
```

viWrite. The **viWrite** function synchronously sends the data pointed to by *buf* to the device specified by *vi*. Only one synchronous write operation can occur at any one time.

```
viWrite(vi, buf, count, retCount);
```

viReadAsync. The **viReadAsync** function asynchronously reads raw data from the session specified by the *vi* parameter and stores the results in the location where *buf* is pointing. This operation normally returns before the transfer terminates. Thus, the operation returns *jobId*, which you can use with either **viTerminate** to abort the operation or with an I/O completion event to identify which asynchronous read operation completed.

```
viReadAsync(vi, buf, count, jobId);
```

viWriteAsync. The **viWriteAsync** function asynchronously sends the data pointed to by *buf* to the device specified by *vi*. This operation normally returns before the transfer terminates. Thus, the operation returns *jobId*, which you can use with either **viTerminate** to abort the operation or with anI/O completion event to identify which asynchronous write operation completed.

```
viWriteAsync(vi, buf, count, jobId);
```

Example: Using Non-Formatted I/O Functions

This example program illustrates using non-formatted I/O functions to communicate with a GPIB device. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter for more information.

```
/*nonfmtio.c
This example program measures the AC voltage on
a multimeter and prints the results. You may
need to change the device address. */

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

void main () {

  ViSession defaultRM, vi;
  char strres [20];
  unsigned long actual;
```

```
/* Open session to GPIB device at address 22 */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::22::INSTR",
  VI_NULL,VI_NULL, &vi);

/* Initialize device */
viWrite(vi, (ViBuf)"*RST\n", 5, &actual);

/* Set up device and take measurement */
viWrite(vi, (ViBuf)"CALC:DBM:REF 50\n", 16,
  &actual);
viWrite(vi, (ViBuf)"MEAS:VOLT:AC? 1, 0.001\n",
  23, &actual);

/* Read results */
viRead(vi, (ViBuf)strres, 20, &actual);

/* NULL terminate the string */
strres[actual]=0;

/* Print results */
printf("Measurement Results: %s\n", strres);

/* Close session */
viClose(vi);
viClose(defaultRM);
}
```

Using Events and Handlers

This section provides guidelines to using events and handlers, including:

- Events and Attributes
- Using the Callback Method
- Using the Queuing Method

Events and Attributes

Events are special occurrences that require attention from your application. Event types include Service Requests (SRQs), interrupts, and hardware triggers. Events will not be delivered unless the appropriate events are enabled.

NOTE

VISA cannot callback to a Visual Basic function. Thus, you can only use the **queuing** mechanism in **viEnableEvent**. There is no way to install a VISA event handler in Visual Basic.

Event Notification

There are two ways you can receive notification that an event has occurred:

- Install an event handler with **viInstallhandler**, and enable one or several events with **viEnableEvent**. If the event was enabled with a handler, the specified event handler will be called when the specified event occurs. This is called a **callback**.

NOTE

VISA cannot callback to a Visual Basic function. This means that you can only use the **VI_QUEUE** mechanism in **viEnableEvent**. There is no way to install a VISA event handler in Visual Basic.

- Enable one or several events with **viEnableEvent** and call the **viWaitOnEvent** function. The **viWaitOnEvent** function will suspend the program execution until the specified event occurs or the specified timeout period is reached. This is called **queuing**.

The queuing and callback mechanisms are suitable for different programming styles. The queuing mechanism is generally useful for non-critical events that do not need immediate servicing. The callback mechanism is useful when immediate responses are needed. These mechanisms work independently of each other, so both can be enabled at the same time. By default, a session is not enabled to receive any events by either mechanism.

The **viEnableEvent** operation can be used to enable a session to respond to a specified event type using either the queuing mechanism, the callback mechanism, or both. Similarly, the **viDisableEvent** operation can be used to disable one or both mechanisms. Because the two methods work independently of each other, one can be enabled or disabled regardless of the current state of the other.

Events that can be Enabled

The following table shows the events that are implemented for Agilent VISA for each resource class, where AP = Access Privilege, RO - Read Only, and RW = Read/Write. Note that some resource classes/events, such as the **SERVANT** class are not implemented by Agilent VISA and are not listed in the following tables.

Once the application has received an event, information about that event can be obtained by using the **viGetAttribute** function on that particular event context. Use the VISA **viReadSTB** function to read the status byte of the service request.

Table 16 Instrument Control (INSTR) Resource Events

VI_EVENT_SERVICE_REQUEST
Notification that a service request was received from the device.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_SERVICE_REQ

VI_EVENT_VXI_SIGP
Notification that a VXIbus signal or VXIbus interrupt was received from the device.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_STOP
VI_ATTR_SIGP_STATUS_ID	The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.	RO	ViUInt16	0 to $FFFF_h$

VI_EVENT_TRIG
Notification that a trigger interrupt was received from the device. For VISA, the only triggers that can be sensed are VXI hardware triggers on the assertion edge (SYNC and ON trigger protocols only).

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_TRIG
VI_ATTR_RECV_TRIG_ID	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1*

* Agilent VISA can also return VI_TRIG_PANEL_IN (exception to the VISA Specification)

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION

Table 16 Instrument Control (INSTR) Resource Events

VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$
VI_ATTR_OPER_NAME	Name of the operation generating the event.		ViString	N/A

VI_EVENT_USB_INTR
Notification that a vendor-specific USB interrupt was received from the device.

Event Attribute	**Description**	**AP**	**Data Type**	**Range**
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_USB_INTR
VI_ATTR_USB_RECV_INTR_ SIZE	Specifies the size of the data that was received from the USB interrupt-IN pipe. This value will never be larger than the sessions value of VI_ATTR_USB_MAX_ INTR_SIZE.	RO	ViUInt16	0 to FFFFh
VI_ATTR_USB_RECV_INTR _DATA	Specifies the actual data that was received from the USB interrupt-IN pipe. Querying this attribute copies the contents of the data to the users buffer. The users buffer must be sufficiently large enough to hold all of the data.	RO	ViBuf	N/A

Table 16 Instrument Control (INSTR) Resource Events

VI_ATTR_STATUS	Specifies the status of the read operation from the USB interrupt-IN pipe. If the device sent more data than the user specified in VI_ATTR_USB_MAX_INTR_SIZE, then this attribute value will contain an error code.	RO	ViStatus	N/A

Table 17 Memory Access (MEMACC) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A

Table 18 GPIB Bus Interface (INTFC) Resource Events

VI_EVENT_GPIB_CIC
Notification that the GPIB controller has gained or lost CIC (controller in charge) status

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_CIC
VI_ATTR_GPIB_RECV_CIC_ STATE	Controller has become controller-in-charge.	RO	ViBoolean	VI_TRUE VI_FALSE

VI_EVENT_GPIB_TALK
Notification that the GPIB controller has been addressed to talk

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_TALK

VI_EVENT_GPIB_LISTEN
Notification that the GPIB controller has been addressed to listen.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_LISTEN

VI_EVENT_CLEAR
Notification that the GPIB controller has been sent a device clear message.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_CLEAR

VI_EVENT_TRIGGER
Notification that a trigger interrupt was received from the interface.

Event Attribute	Description	AP	Data Type	Range

Table 18 GPIB Bus Interface (INTFC) Resource Events

VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_TRIG
VI_ATTR_RECV_TRIG_ID	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	ViInt16	VI_TRIG_SW

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_ COMPLETION
VI_ATTR_STATUS	Return code of the asynch-ronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of buffer used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$
VI_ATTR_OPER_NAME	The name of the operation generating the event.	RO	ViString	N/A

Table 19 VXI Mainframe Backplane (BACKPLANE) Resource Events

VI_EVENT_TRIG
Notification that a trigger interrupt was received from the backplane. For VISA, the only triggers that can be sensed are VXI hardware triggers on the assertion edge (SYNC and ON trigger protocols only).

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_TRIG

Table 19 VXI Mainframe Backplane (BACKPLANE) Resource Events

VI_ATTR_RECV_TRIG_ID	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_EVENT_VXI_VME_SYSFAIL Notification that the VXI/VME SYSFAIL* line has been asserted.				
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_ SYSFAIL
VI_EVENT_VXI_VME_SYSRESET Notification that the VXI/VME SYSRESET* line has been reset				
Event Attributes	**Description**	**AP**	**Data Type**	**Range**
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_ SYSRESET

Table 20 TCPIP Socket (SOCKET) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$

Table 20 TCPIP Socket (SOCKET) Resource Event

VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A

Example: Reading Event Attributes

Once you have decided which attribute to check, you can read the attribute using the **viGetAttribute** function. The following example shows one way you could check which trigger line fired when the VI_EVENT_TRIG event was delivered.

Note that the *context* parameter is either the event *context* passed to your event handler, or the *outcontext* specified when doing a wait on event. See “VISA Attributes” in this chapter for more information on reading attribute states.

```
ViInt16 state;
.
.
viGetAttribute(context, VI_ATTR_RECV_TRIG_ID,
   &state)
```

Using the Callback Method

The callback method of event notification is used when an immediate response to an event is required. To use the callback method for receiving notification that an event has occurred, you must do the following.

- Install an event handler with the **viInstallHandler** function
- Enable one or several events with the **viEnableEvent** function

When the enabled event occurs, the installed event handler is called.

Example: Using the Callback Method

This example shows one way you can use the callback method.

```
ViStatus _VI_FUNCH my_handler (ViSession vi,
  ViEventType eventType, ViEvent context, ViAddr
  usrHandle) {

/* your event handling code here */

return VI_SUCCESS;

}
main(){
ViSession vi;
ViAddr addr=0;
.
.
viInstallHandler(vi, VI_EVENT_SERVICE_REQ,
  my_handler, addr);
viEnableEvent(vi, VI_EVENT_SERVICE_REQ,
  VI_HNDLR, VI_NULL);
.
/* your code here */
.
viDisableEvent(vi, VI_EVENT_SERVICE_REQ,
  VI_HNDLR);
viUninstallHandler(vi, VI_EVENT_SERVICE_REQ,
  my_handler, addr);
.
}
```

Installing Handlers

VISA allows applications to install multiple handlers for an event type on the same session. Multiple handlers can be installed through multiple invocations of the **viInstallHandler** operation, where each invocation adds to the previous list of handlers.

If more than one handler is installed for an event type, each of the handlers is invoked on every occurrence of the specified event(s). VISA specifies that the handlers are invoked in Last In First Out (LIFO) order. Use the following function when installing an event handler:

```
viInstallHandler(vi, eventType, handler,
  userHandle);
```

These parameters are defined as follows.

Table 21 Parameters Used to Install a Handler

Parameter	Description
vi	The session on which the handler will be installed.
eventType	The event type that will activate the handler.
handler	The name of the handler to be called.
userHandle	A user value that uniquely identifies the handler for the specified event type.

The *userHandle* parameter allows you to assign a value to be used with the *handler* on the specified session. Thus, you can install the same handler for the same event type on several sessions with different *userHandle* values. The same handler is called for the specified event type.

However, the value passed to *userHandle* is different. Therefore the handlers are uniquely identified by the combination of the *handler* and the *userHandle*. This may be useful when you need a different handling method depending on the *userHandle*.

Example: Installing an Event Handler

This example shows how to install an event handler to call *my_handler* when a Service Request occurs. Note that VI_EVENT_SERVICE_REQ must also be an enabled event with the **viEnableEvent** function for the service request event to be delivered.

```
viInstallHandler(vi, VI_EVENT_SERVICE_REQ,
  my_handler, addr);
```

Use the **viUninstallHandler** function to uninstall a specific handler, or you can use wildcards (VI_ANY_HNDLR in the *handler* parameter) to uninstall groups of handlers. See **viUninstallHandler** in the *VISA Online Help* for more details on this function.

Writing the Handler

The *handler* installed needs to be written by the programmer. The event handler typically reads an associated attribute and performs some sort of action. See the event handler in the example program later in this section.

Enabling Events

Before an event can be delivered, it must be enabled using the **viEnableEvent** function. This function causes the application to be notified when the enabled event has occurred, where the parameters are:

```
viEnableEvent(vi, eventType, mechanism,
  context);
```

Using VI_QUEUE in the *mechanism* parameter specifies a queuing method for the events to be handled. If you use both VI_QUEUE and one of the mechanisms listed above, notification of events will be sent to both locations. See the next subsection for information on the queuing method.

Table 22 Description of Parameters Used to Install a Handler

Parameter	Description
vi	The session on which the handler will be installed.
eventType	The type of event to enable.

Table 22 Description of Parameters Used to Install a Handler

mechanism	The mechanism by which the event will be enabled. It can be enabled in several different ways. You can use VI_HNDLR in this parameter to specify that the installed handler will be called when the event occurs. Use VI_SUSPEND_HNDLR in this parameter, which puts the events in a queue and waits to call the installed handlers until **viEnableEvent** is called with VI_HNDLR specified in the mechanism parameter. When **viEnableEvent** is called with VI_HNDLR specified, the handler for each queued event will be called.
context	Not used in VISA 1.0. Use VI_NULL.

Example: Enabling a Hardware Trigger Event

This example illustrates enabling a hardware trigger event.

```
viInstallHandler(vi, VI_EVENT_TRIG,
   my_handler,&addr);
viEnableEvent(vi, VI_EVENT_TRIG, VI_HNDLR,
   VI_NULL);
```

The VI_HNDLR mechanism specifies that the handler installed for VI_EVENT_TRIG will be called when a hardware trigger occurs.

If you specify VI_ALL_ENABLE_EVENTS in the *eventType* parameter, all events that have previously been enabled on the specified session will be enabled for the *mechanism* specified in this function call.

Use the **viDisableEvent** function to stop servicing the event specified.

Example: Trigger Callback

This example program installs an event handler and enables the trigger event. When the event occurs, the installed event handler is called. This program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter for more information.

This example program is installed on your system in the `Samples` subdirectory on Windows environments. See *VISA Library Information* in the *VISA Online Help* for locations of example programs.

```
/* evnthdlr.c
This example program illustrates installing an
event handler to be called when a trigger
interrupt occurs. Note that you may need to
change the address. */

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

/* trigger event handler */
ViStatus _VI_FUNCH myHdlr(ViSession vi,
      ViEventType eventType, ViEvent ctx, ViAddr
      userHdlr){
   ViInt16 trigId;

/* make sure it is a trigger event */
if(eventType!=VI_EVENT_TRIG){
   /* Stray event, so ignore */
   return VI_SUCCESS;
}

/* print the event information */
printf("Trigger Event Occurred!\n");
printf("...Original Device Session = %ld\n",
   vi);

/* get the trigger that fired */
viGetAttribute(ctx, VI_ATTR_RECV_TRIG_ID,
   &trigId);
printf("Trigger that fired: ");
switch(trigId){
   case VI_TRIG_TTL0:
      printf("TTL0");
      break;
   default:
      printf("<other 0x%x>", trigId);
      break;
}
```

```
printf("\n");

return VI_SUCCESS;
}

void main(){
ViSession defaultRM,vi;

/* open session to VXI device */
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL,
   VI_NULL, &vi);

/* select trigger line TTL0 */
viSetAttribute(vi, VI_ATTR_TRIG_ID,
   VI_TRIG_TTL0);
/* install the handler and enable it */
viInstallHandler(vi, VI_EVENT_TRIG, myHdlr,
   (ViAddr)10);
viEnableEvent(vi, VI_EVENT_TRIG, VI_HNDLR,
   VI_NULL);
/* fire trigger line, twice */
viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
viAssertTrigger(vi, VI_TRIG_PROT_SYNC);

/* unenable and uninstall the handler */
viDisableEvent(vi, VI_EVENT_TRIG, VI_HNDLR);

viUninstallHandler(vi, VI_EVENT_TRIG, myHdlr,
   (ViAddr)10);

/* close the sessions */
viClose(vi);
viClose(defaultRM);
}
```

Example: SRQ Callback

This program installs an event handler and enables an SRQ event. When the event occurs, the installed event handler is called. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter for more information.

This program is installed on your system in the `Samples` subdirectory on Windows environments. See *VISA Library Information* in the *VISA Online Help* for locations of example programs.

```
/* srqhdlr.c
This example program illustrates installing an
event handler to be called when an SRQ interrupt
occurs. Note that you may need to change the
address. */

#include <visa.h>
#include <stdio.h>
#if defined (_WIN32)
  #include <windows.h> /* for Sleep() */
  #define YIELD Sleep( 10 )
#elif defined (_BORLANDC_)
  #include <windows.h>  /* for Yield() */
  #define YIELD Yield()
#elif defined (_WINDOWS)
  #include <io.h>      /* for _wyield */
  #define YIELD   _wyield()
#else
  #include <unistd.h>
  #define YIELD sleep (1)
#endif

int srqOccurred;

/* trigger event handler */
ViStatus _VI_FUNCH mySrqHdlr(ViSession vi,
   ViEventType
eventType, ViEvent ctx, ViAddr userHdlr){

  ViUInt16 statusByte;

  /* make sure it is an SRQ event */
  if(eventType!=VI_EVENT_SERVICE_REQ){
    /* Stray event, so ignore */
    printf( "\nStray event of type 0x%lx\n",
```

```
    eventType );
    return VI_SUCCESS;
    }

    /* print the event information */
    printf("\nSRQ Event Occurred!\n");
    printf("...Original Device Session = %ld\n",
      vi);

    /* get the status byte */
    viReadSTB(vi, &statusByte);
    printf("...Status byte is 0x%x\n",
      statusByte);

    srqOccurred = 1;
    return VI_SUCCESS;
}

void main(){
    ViSession defaultRM,vi;
    long count;

    /* open session to message based VXI device */
    viOpenDefaultRM(&defaultRM);
    viOpen(defaultRM, "GPIB-VXI0::24::INSTR",
      VI_NULL, VI_NULL, &vi);

    /* Enable command error events */
    viPrintf( vi, "*ESE 32\n" );

    /* Enable event register interrupts */
    viPrintf( vi, "*SRE 32\n" );

    /* install the handler and enable it */
    viInstallHandler(vi, VI_EVENT_SERVICE_REQ,
      mySrqHdlr,
    (ViAddr)10);
    viEnableEvent(vi, VI_EVENT_SERVICE_REQ,
      VI_HNDLR, VI_NULL);

    srqOccurred = 0;
```

```
/* Send a bogus command to the message-based
device to cause an SRQ. Note: 'IDN' causes the
error -- 'IDN?' is the correct syntax */
viPrintf( vi, "IDN\n" );

/* Wait a while for the SRQ to be generated and
for the handler to be called. Print something
while we wait */

printf("Waiting for an SRQ to be generated.");
for (count = 0 ; (count < 10) &&
  (srqOccurred == 0);count++) {
  long count2 = 0;
  printf( "." );
  while ( (count2++ < 100) && (srqOccurred ==0)
 ){YIELD;
 }
}
printf( "\n" );

/* disable and uninstall the handler */
viDisableEvent(vi, VI_EVENT_SERVICE_REQ,
  VI_HNDLR);
viUninstallHandler(vi, VI_EVENT_SERVICE_REQ,
  mySrqHdlr, (ViAddr)10);

/* Clean up - do not leave device in error
state */
viPrintf( vi, "*CLS\n" );

/* close the sessions */
viClose(vi);
viClose(defaultRM);

printf( "End of program\n" );}
```

Using the Queuing Method

The queuing method is generally used when an immediate response from your application is not needed. To use the queuing method for receiving notification that an event has occurred, you must do the following:

- Enable one or several events with the **viEnableEvent** function.
- When ready to query, use the **viWaitOnEvent** function to check for queued events.

If the specified event has occurred, the event information is retrieved and the program returns immediately. If the specified event has not occurred, the program suspends execution until a specified event occurs or until the specified timeout period is reached.

Example: Using the Queuing Method

This example program shows one way you can use the queuing method.

```
main();
ViSession vi;
ViEventType eventType;
ViEvent event;
.
.
viEnableEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_QUEUE, VI_NULL);
.
.
viWaitOnEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_TMO_INFINITE, &eventType, &event);
.
.
viClose(event);
viDisableEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_QUEUE);
}
```

Enabling Events

Before an event can be delivered, it must be enabled using the **viEnableEvent** function:

```
viEnableEvent(vi, eventType, mechanism,
   context);
```

These parameters are defined as follows:

Table 23 Descriptions of Parameters Used to Enable Events

Parameter	Description
vi	The session the handler will be installed on.
eventType	The type of event to enable.
mechanism	The mechanism by which the event will be enabled. Specify VI_QUEUE to use the queuing method.
context	Not used in VISA 1.0. Use VI_NULL.

When you use VI_QUEUE in the *mechanism* parameter, you are specifying that the events will be put into a queue. Then, when a **viWaitOnEvent** function is invoked, the program execution will suspend until the enabled event occurs or the timeout period specified is reached. If the event has already occurred, the **viWaitOnEvent** function will return immediately.

Example: Enabling a Hardware Trigger Event

This example illustrates enabling a hardware trigger event.

```
viEnableEvent(vi, VI_EVENT_TRIG, VI_QUEUE,
   VI_NULL);
```

The VI_QUEUE mechanism specifies that when an event occurs, it will go into a queue. If you specify VI_ALL_ENABLE_EVENTS in the *eventType* parameter, all events that have previously been enabled on the specified session will be enabled for the *mechanism* specified in this function call. Use the **viDisableEvent** function to stop servicing the event specified.

Wait on the Event

When using the **viWaitOnEvent** function, specify the session, the event type to wait for, and the timeout period to wait:

```
viWaitOnEvent(vi, inEventType, timeout,
   outEventType, outContext);
```

The event must have previously been enabled with VI_QUEUE specified as the *mechanism* parameter.

Example: Wait on Event for SRQ

This example shows how to install a wait on event for service requests.

```
viEnableEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_QUEUE, VI_NULL);
viWaitOnEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_TMO_INFINITE, &eventType, &event);
.
.
viDisableEvent(vi, VI_EVENT_SERVICE_REQ,
   VI_QUEUE);
```

Every time a wait on event is invoked, an event context object is created. Specifying VI_TMO_INFINITE in the *timeout* parameter indicates that the program execution will suspend indefinitely until the event occurs. To clear the event queue for a specified event type, use the **viDiscardEvents** function.

Example: Trigger Event Queuing

This program enables the trigger event in a queuing mode. When the **viWaitOnEvent** function is called, the program will suspend operation until the trigger line is fired or the timeout period is reached. Since the trigger lines were already fired and the events were put into a queue, the function will return and print the trigger line that fired.

This program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See “Trapping Errors” in this chapter for more information.

This example program is installed on your system in the `Samples` subdirectory on Windows environments. See *VISA Library Information* in the *VISA Online Help* for locations of example programs.

```
/* evntqueu.c
This example program illustrates enabling an
event queue using viWaitOnEvent. Note that you
must change the device address. */

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

void main(){
  ViSession defaultRM,vi;
  ViEventType eventType;
  ViEvent eventVi;
  ViStatus err;
  ViInt16 trigId;

  /* open session to VXI device */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL,
    VI_NULL, &vi);

  /* select trigger line TTL0 */
  viSetAttribute(vi, VI_ATTR_TRIG_ID,
    VI_TRIG_TTL0);

  /* enable the event */
  viEnableEvent(vi, VI_EVENT_TRIG, VI_QUEUE,
    VI_NULL);

  /* fire trigger line, twice */
  viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
  viAssertTrigger(vi, VI_TRIG_PROT_SYNC);

  /* Wait for the event to occur */
  err=viWaitOnEvent(vi, VI_EVENT_TRIG, 10000,
    &eventType, &eventVi);
  if(err==VI_ERROR_TMO){
    printf("Timeout Occurred! Event not
    received.\n");
  return;
  }
```

```
  /* print the event information */
  printf("Trigger Event Occurred!\n");
  printf("...Original Device Session = %ld\n",
    vi);

  /* get trigger that fired */
  viGetAttribute(eventVi, VI_ATTR_RECV_TRIG_ID,
    &trigId);
  printf("Trigger that fired: ");
  switch(trigId){
  case VI_TRIG_TTL0:
    printf("TTL0");
    break;
  default:
    printf("<other 0x%x>",trigId);
    break;
  }
  printf("\n");

  /* close the context before continuing */
  viClose(eventVi);

  /* get second event */
  err=viWaitOnEvent(vi, VI_EVENT_TRIG, 10000,
    &eventType, &eventVi);
  if(err==VI_ERROR_TMO){
    printf("Timeout Occurred! Event not
      received.\n");
    return;
  }
  printf("Got second event\n");

  /* close the context before continuing */
  viClose(eventVi);

  /* disable event */
  viDisableEvent(vi, VI_EVENT_TRIG, VI_QUEUE);

  /* close the sessions */
  viClose(vi);
  viClose(defaultRM);
}
```

Trapping Errors

This section provides guidelines for trapping errors, including:

- Trapping Errors
- Exception Events

Trapping Errors

The example programs in this guide show specific VISA functionality and do not include error trapping. Error trapping, however, is good programming practice and is recommended in all your VISA application programs. To trap VISA errors you must check for VI_SUCCESS after each VISA function call.

If you want to ignore WARNINGS, you can test to see if **err** is less than (**<**) VI_SUCCESS. Since WARNINGS are greater than VI_SUCCESS and ERRORS are less than VI_SUCCESS, **err_handler** would only be called when the function returns an ERROR. For example:

```
if(err < VI_SUCCESS) err_handler (vi, err);
```

Example: Checking for VI_SUCCESS

This example illustrates checking for VI_SUCCESS. If VI_SUCCESS is not returned, an error handler (written by the programmer) is called. This must be done with each VISA function call.

```
ViStatus err;
.
.
err=viPrintf(vi, "*RST\n");
if (err < VI_SUCCESS) err_handler(vi, err);
.
.
```

Example: Printing Error Code

The following error handler prints a user-readable string describing the error code passed to the function:

```
void err_handler(ViSession vi, ViStatus err){

  char err_msg[1024]={0};
  viStatusDesc (vi, err, err_msg);
  printf ("ERROR = %s\n", err_msg);
  return;
}
```

Example: Checking Instrument Errors

When programming instruments, it is good practice to check the instrument to ensure there are no instrument errors after each instrument function. This example uses a SCPI command to check a specific instrument for errors.

```
void system_err(){

  ViStatus err;
  char buf[1024]={0};
  int err_no;

  err=viPrintf(vi, "SYSTEM:ERR?\n");
  if (err < VI_SUCCESS) err_handler (vi, err);

  err=viScanf (vi, "%d%t", &err_no, &buf);
  if (err < VI_SUCCESS) err_handler (vi, err);

  while (err_no >0){
    printf ("Error Found: %d,%s\n", err_no,
      buf);
    err=viScanf (vi, "%d%t", &err_no, &buf);
  }
  err=viFlush(vi, VI_READ_BUF);
  if (err < VI_SUCCESS) err_handler (vi, err);

  err=viFlush(vi, VI_WRITE_BUF);
  if (err < VI_SUCCESS) err_handler (vi, err);
}
```

Exception Events

An alternative to trapping VISA errors by checking the return status after each VISA call is to use the VISA **exception event**. On sessions where an exception event handler is installed and

VI_EVENT_EXCEPTION is enabled, the exception event handler is called whenever an error occurs while executing an operation.

Exception Handling Model

The exception-handling model follows the event-handling model for callbacks, and it uses the same operations as those used for general event handling. For example, an application calls **viInstallHandler** and **viEnableEvent** to enable exception events. The exception event is like any other event in VISA, except that the queueing and suspended handler mechanisms are not allowed.

When an error occurs for a session operation, the exception handler is executed synchronously. That is, the operation that caused the exception blocks until the exception handler completes its execution. The exception handler is executed in the context of the same thread that caused the exception event.

When invoked, the exception handler can check the error condition and instruct the exception operation to take a specific action. It can instruct the exception operation to continue normally (by returning VI_SUCCESS) or to not invoke any additional handlers in the case of handler nesting (by returning VI_SUCCESS_NCHAIN).

As noted, an exception operation blocks until the exception handler execution is completed. However, an exception handler sometimes may prefer to terminate the program prematurely without returning the control to the operation generating the exception. VISA does not preclude an application from using a platform-specific or language-specific exception handling mechanism from within the VISA exception handler.

For example, the C++ try/catch block can be used in an application in conjunction with the C++ throw mechanism from within the VISA exception handler. When using the C++

try/catch/throw or other exception-handling mechanisms, the control will not return to the VISA system. This has several important repercussions:

1 If multiple handlers were installed on the exception event, the handlers that were not invoked prior to the current handler will not be invoked for the current exception.

2 The exception context will not be deleted by the VISA system when a C++ exception is used. In this case, the application should delete the exception context as soon as the application has no more use for the context, before terminating the session. An application should use the **viClose** operation to delete the exception context.

3 Code in any operation (after calling an exception handler) may not be called if the handler does not return. For example, local allocations must be freed before invoking the exception handler, rather than after it.

One situation in which an exception event will not be generated is in the case of asynchronous operations. If the error is detected after the operation is posted (i.e., once the asynchronous portion has begun), the status is returned normally via the I/O completion event.

However, if an error occurs before the asynchronous portion begins (i.e., the error is returned from the asynchronous operation itself), then the exception event will still be raised. This deviation is due to the fact that asynchronous operations already raise an event when they complete, and this I/O completion event may occur in the context of a separate thread previously unknown to the application. In summary, a single application event handler can easily handle error conditions arising from both exception events and failed asynchronous operations.

Using the VI_EVENT_EXCEPTION Event

You can use the VI_EVENT_EXCEPTION event as notification that an error condition has occurred during an operation invocation. The following table describes the VI_EVENT_EXCEPTION event attributes.

Table 24 VI_EVENT_EXCEPTION Event Attributes.

Attribute Name	Access Privilege		Data Type	Range	Default
VI_ATTR_EVENT_TYPE	RO	Global	ViEventType	VI_EVENT_EXCEPTION	N/A
VI_ATTR_STATUS	RO	Global	ViStatus	N/A	N/A
VI_ATTR_OPER_NAME	RO	Global	ViString	N/A	N/A

Example: Exception Events

```
/* This is an example of how to use exception
events to trap VISA errors. An exception event
handler must be installed and exception events
enabled on all sessions where the exception
handler is used.*/

#include <stdio.h>
#include <visa.h>
ViStatus __stdcall myExceptionHandler (
  ViSession vi,
  ViEventType eventType,
  ViEvent context,
  ViAddr usrHandle
) {
  ViStatus exceptionErrNbr;
  char     nameBuffer[256];
  ViString functionName = nameBuffer;
  char     errStrBuffer[256];
  /* Get the error value from the exception
     context */
  viGetAttribute( context, VI_ATTR_STATUS,
    &exceptionErrNbr );
/* Get the function name from the exception
   context */
   viGetAttribute( context, VI_ATTR_OPER_NAME,
     functionName );
errStrBuffer[0] = 0;
  viStatusDesc( vi, exceptionErrNbr,
    errStrBuffer );
```

```
  printf("ERROR: Exception Handler reports\n"
    "(%s)\n","VISA function '%s' failed with
    error 0x%lx\n", "functionName,
    exceptionErrNbr, errStrBuffer );
  return VI_SUCCESS;
}
void main(){
  ViStatus  status;
  ViSession drm;
  ViSession vi;
  ViAddr     myUserHandle = 0;

  status = viOpenDefaultRM( &drm );
  if ( status < VI_SUCCESS ) {
    printf( "ERROR: viOpenDefaultRM failed with
     error = 0x%lx\n", status );
    return;
  }

/* Install the exception handler and enable
   events for it */
   status = viInstallHandler(drm,
     VI_EVENT_EXCEPTION, myExceptionHandler,
     myUserHandle);
   if ( status < VI_SUCCESS )
{
      printf( "ERROR: viInstallHandler failed
        with error 0x%lx\n", status );
   }

status = viEnableEvent(drm, VI_EVENT_EXCEPTION,
  VI_HNDLR, VI_NULL);
if ( status < VI_SUCCESS ) {
  printf( "ERROR: viEnableEvent failed with
    error 0x%lx\n", status );
   }

/* Generate an error to demonstrate that the
   handler will be called */
   status = viOpen( drm, "badVisaName", NULL,
   NULL, &vi );
   if ( status < VI_SUCCESS ) {
```

```
        printf("ERROR: viOpen failed with error
          0x%lx\n"
          "Exception Handler should have been
          called\n"
         "before this message was printed.\n",status
          );
      }
   }
```

Using Locks

In VISA, applications can open multiple sessions to a VISA resource simultaneously. Applications can, therefore, access a VISA resource concurrently through different sessions. However, in certain cases, applications accessing a VISA resource may want to restrict other applications from accessing that resource.

Lock Functions

For example, when an application needs to perform successive write operations on a resource, the application may require that, during the sequence of writes, no other operation can be invoked through any other session to that resource. For such circumstances, VISA defines a locking mechanism that restricts access to resources.

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

If a VISA resource is not locked by any of its sessions, all sessions have full privilege to invoke any operation and update any global attributes. Sessions are *not* required to have locks to invoke operations or update global attributes. However, if some other session has already locked the resource, attempts to update global attributes or invoke certain operations will fail.

See descriptions of the individual VISA functions in the *VISA Online Help* to determine which would fail when a resource is locked.

viLock/viUnlock Functions

The VISA **viLock** function is used to acquire a lock on a resource.

```
viLock(vi, lockType, timeout, requestedKey,
  accessKey);
```

The VI_ATTR_RSRC_LOCK_STATE attribute specifies the current locking state of the resource on the given session, which can be either VI_NO_LOCK, VI_EXCLUSIVE_LOCK, or VI_SHARED_LOCK.

The VISA **viUnlock** function is then used to release the lock on a resource. If a resource is locked and the current session does not have the lock, the error VI_ERROR_RSRC_LOCKED is returned.

VISA Lock Types

VISA defines two different types of locks: **Exclusive Lock** and **Shared Lock**.

Exclusive Lock - A session can lock a VISA resource using the lock type VI_EXCLUSIVE_LOCK to get exclusive access privileges to the resource. This exclusive lock type excludes access to the resource from all other sessions.

If a session has an exclusive lock, other sessions cannot modify global attributes or invoke operations on the resource. However, the other sessions *can* still get atttributes.

Shared Lock - A session can share a lock on a VISA resource with other sessions by using the lock type VI_SHARED_LOCK. Shared locks in VISA are similar to exclusive locks in terms of access privileges, but can still be shared between multiple sessions.

If a session has a shared lock, other sessions that share the lock can also modify global attributes and invoke operations on the resource (of course, unless some other session has a previous exclusive lock on that resource). A session that does not share the lock will lack these capabilities.

Locking a resource restricts access from other sessions, and in the case where an exclusive lock is acquired, ensures that operations do not fail because other sessions have acquired a lock on that resource. Thus, locking a resource prevents other, subsequent sessions from acquiring an exclusive lock on that resource. Yet, when multiple sessions have acquired a shared lock, VISA allows one of the sessions to acquire an exclusive lock along with the shared lock it is holding.

Also, VISA supports nested locking. That is, a session can lock the same VISA resource multiple times (for the same lock type) via multiple invocations of the **viLock** function. In such a case, unlocking the resource requires an equal number of invocations of the **viUnlock** function. Nested locking is explained in detail later in this section.

Some VISA operations may be permitted even when there is an exclusive lock on a resource, or some global attributes may not be read when there is any kind of lock on the resource. These exceptions, when applicable, are mentioned in the descriptions of the individual VISA functions and attributes.

See the *VISA Online Help* for descriptions of individual functions to determine which are applicable for locking and which are not restricted by locking.

Example: Exclusive Lock

This example shows a session gaining an exclusive lock to perform the **viPrintf** and **viScanf** VISA operations on a GPIB device. It then releases the lock via the **viUnlock** function.

```
/* lockexcl.c
This example program queries a GPIB device for
an identification string and prints the results.
Note that you may need to change the address. */

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

void main () {

  ViSession defaultRM, vi;
  char buf [256] = {0};

  /* Open session to GPIB device at address 22 */
  viOpenDefaultRM (&defaultRM);
  viOpen (defaultRM, "GPIB0::22::INSTR",
    VI_NULL,VI_NULL, &vi);
```

```
/* Initialize device */
viPrintf (vi, "*RST\n");

/* Make sure no other process or thread does
anything to this resource between viPrintf and
viScanf calls */

viLock (vi, VI_EXCLUSIVE_LOCK, 2000, VI_NULL,
  VI_NULL);

/* Send an *IDN? string to the device */
viPrintf (vi, "*IDN?\n");

/* Read results */
viScanf (vi, "%t", &buf);

/* Unlock this session so other processes and
threads can use it */
viUnlock (vi);

/* Print results */
printf ("Instrument identification string:
  %s\n", buf);

/* Close session */
viClose (vi);
viClose (defaultRM);}
```

Example: Shared Lock

This example shows a session gaining a shared lock with the *accessKey* called **lockkey**. Other sessions can now use this *accessKey* in the *requestedKey* parameter of the **viLock** function to share access on the locked resource. This example then shows the original session acquiring an exclusive lock while maintaining its shared lock.

When the session holding the exclusive lock unlocks the resource via the **viUnlock** function, all the sessions sharing the lock again have all the access privileges associated with the shared lock.

```
/* lockshr.c
This example program queries a GPIB device for
an identification string and prints the results.
Note that you must change the address. */

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

void main () {

  ViSession defaultRM, vi;
  char buf [256] = {0};
  char lockkey [256] = {0};

  /* Open session to GPIB device at address 22 */
  viOpenDefaultRM (&defaultRM);
  viOpen (defaultRM, "GPIB0::22::INSTR",
    VI_NULL,VI_NULL,&vi);

  /* acquire a shared lock so only this process
  and processes that we know about can access
  this resource */
  viLock (vi, VI_SHARED_LOCK, 2000, VI_NULL,
    lockkey);

  /* at this time, we can make 'lockkey'
  available to other processes that we know
  about. This can be done with shared memory or
  other inter-process communication methods.
  These other processes can then call
  "viLock(vi,VI_SHARED_LOCK, 2000, lockkey,
  lockkey)" and they will also have access to
  this resource.  */

  /* Initialize device */
  viPrintf (vi, "*RST\n");

  /* Make sure no other process or thread does
  anything to this resource between the
  viPrintf() and the viScanf()calls Note: this
  also locks out the processes with which we
  shared our 'shared lock' key.  */

  viLock (vi, VI_EXCLUSIVE_LOCK, 2000,
    VI_NULL,VI_NULL);
```

```
  /* Send an *IDN? string to the device */
  viPrintf (vi, "*IDN?\n");

  /* Read results */
  viScanf (vi, "%t", &buf);

  /* unlock this session so other processes and
  threads can use it */
  viUnlock (vi);

  /* Print results */
  printf ("Instrument identification string:
    %s\n", buf);

  /* release the shared lock also*/
  viUnlock (vi);

  /* Close session */
  viClose (vi);
  viClose (defaultRM);
}
```

4
Programming via GPIB and VXI

VISA supports three interfaces you can use to access GPIB and VXI instruments: GPIB, VXI, and GPIB-VXI.

This chapter provides information to program GPIB and VXI devices via the GPIB, VXI or GPIB-VXI interfaces, including:

- GPIB and VXI Interfaces Overview
- Using High-Level Memory Functions
- Using Low-Level Memory Functions
- Using High/Low-Level Memory I/O Methods
- Using the Memory Access Resource
- Using VXI-Specific Attributes

See Chapter 3, *"Programming with VISA"* for general information on VISA programming for the GPIB, VXI, and GPIB-VXI interfaces. See the *VISA Online Help* for information on the specific VISA functions.

GPIB and VXI Interfaces Overview

This section provides an overview of the GPIB, GPIB-VXI, and VXI interfaces, including:

- General Interface Information
- GPIB Interfaces Overview
- VXI Interfaces Overview

General Interface Information

VISA supports three interfaces you can use to access instruments or devices: GPIB, VXI, and GPIB-VXI. The GPIB interface can be used to access VXI instruments via a Command Module. In addition, the VXI backplane can be directly accessed with the VXI or GPIB-VXI interfaces.

What is an I/O Interface?

An **I/O interface** can be defined as both a hardware interface and as a software interface. The *IO Config* utility is used to associate a unique interface name with a hardware interface. The IO Libraries use a **VISA Interface Name** to identify an interface. This information is passed in the parameter string of the **viOpen** function call in a VISA program.

IO Config assigns a VISA Interface Name to the interface hardware, and other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for IO Config information.

VXI Device Types

When using GPIB-VXI or VXI interfaces to directly access the VXI backplane (in the VXI mainframe), you must know whether you are programming a message-based or a register-based VXI device (instrument).

A **message-based VXI device** has its own processor that allows it to interpret high-level commands such as Standard Commands for Programmable Instruments (SCPI). When using

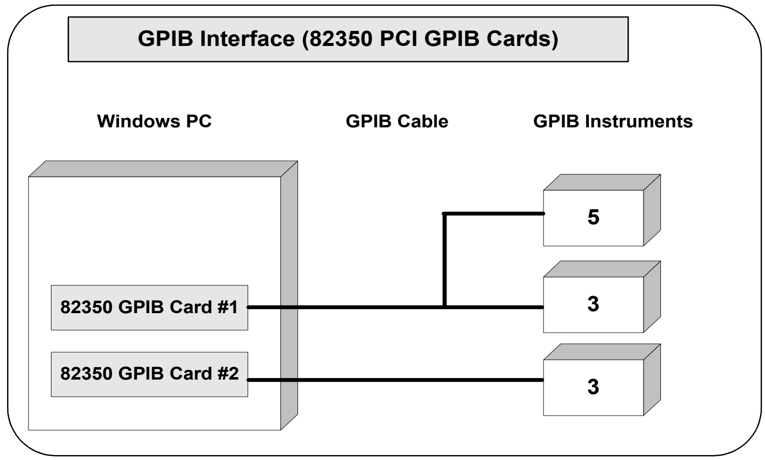
VISA, you can place the SCPI command within your VISA output function call. Then, the message-based device interprets the SCPI command. In this case you can use the VISA formatted I/O or non-formatted I/O functions and program the message-based device as you would a GPIB device.

However, if the message-based device has shared memory, you can access the device's shared memory by doing register peeks and pokes. VISA provides two different methods you can use to program directly to the registers: high-level memory functions or low-level memory functions.

A **register-based VXI device** typically does not have a processor to interpret high-level commands. Therefore, the device must be programmed with register peeks and pokes directly to the device's registers. VISA provides two different methods you can use to program register-based devices: high-level memory functions or low-level memory functions.

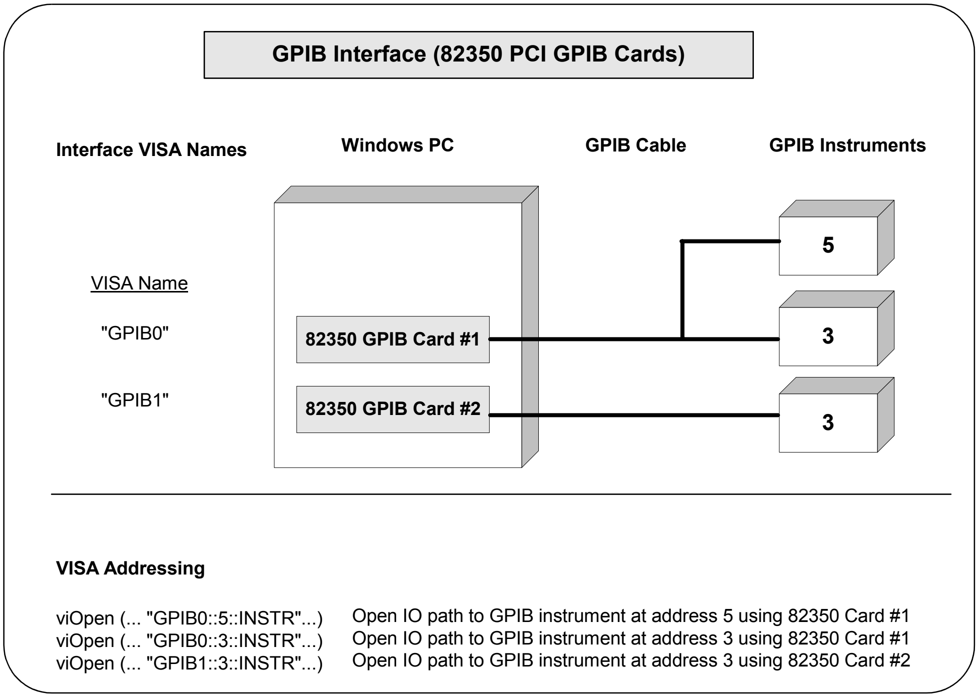
GPIB Interfaces Overview

As shown in the following figure, a typical GPIB interface consists of a Windows PC with one or more GPIB cards (PCI and/or ISA) cards installed in the PC, and one or more GPIB instruments connected to the GPIB cards via GPIB cable. I/O communication between the PC and the instruments is via the GPIB cards and the GPIB cable. The following figure shows GPIB instruments at addresses 3 and 5.



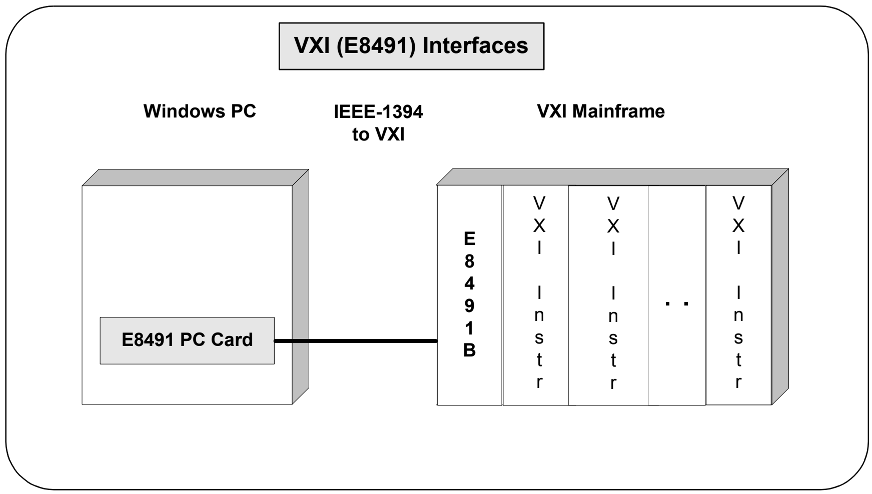
Example: GPIB (82350) Interface

The GPIB interface system in the following figure consists of a Windows PC with two 82350 GPIB cards connected to three GPIB instruments via GPIB cables. For this system, the IO Config utility has been used to assign GPIB card #1 a VISA name of **GPIB0** and to assign GPIB card #2 a VISA name of **GPIB1**. VISA addressing is as shown in the figure.


GPIB Interface (82350 PCI GPIB Cards)
Interface VISA Names
Windows PC
GPIB Cable
GPIB Instruments
VISA Name
"GPIB0"
"GPIB1"
82350 GPIB Card #1
82350 GPIB Card #2
5
3
3
VISA Addressing
viOpen (... "GPIB0::5::INSTR"...)
viOpen (... "GPIB0::3::INSTR"...)
viOpen (... "GPIB1::3::INSTR"...)
Open IO path to GPIB instrument at address 5 using 82350 Card #1
Open IO path to GPIB instrument at address 3 using 82350 Card #1
Open IO path to GPIB instrument at address 3 using 82350 Card #2

VXI Interfaces Overview

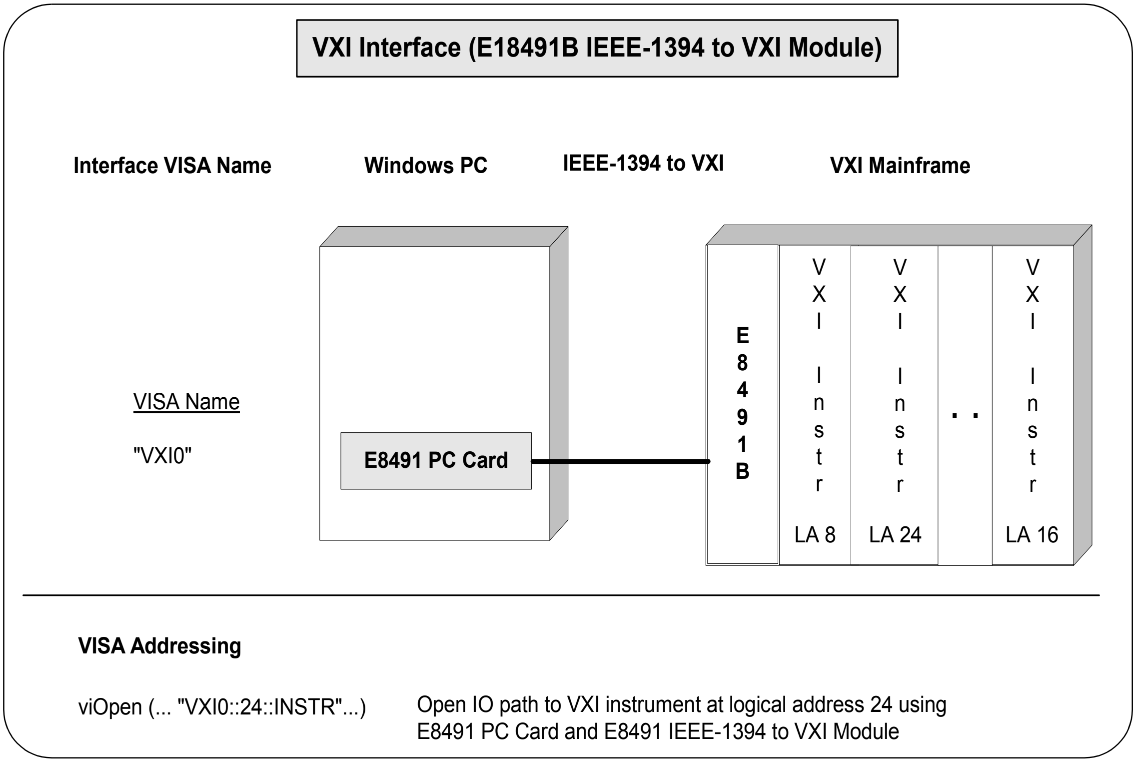
As shown in the following figure, a typical VXI (E8491) interface consists of an E8491 PC Card in a Windows PC that is connected to an E8491B IEEE-1394 Module in a VXI mainframe via an IEEE-1394 to VXI cable. The VXI mainframe also includes one or more VXI instruments.



Example: VXI (E8491B) Interfaces

The VXI interface system in the following figure consists of a Windows PC with an E8491 PC card that connects to an E8491B IEEE-1394 to VXI Module in a VXI Mainframe. For this system, the three VXI instruments shown have logical addresses 8, 16, and 24. The IO Config utility has been used to assign the E8491 PC card a VISA name of **VXI0**. VISA addressing is as shown in the figure.

For information on the E8491B module, see the *Agilent E8491B User's Guide*. For information on VXI instruments, see the applicable *VXI Instrument User's Guide*.



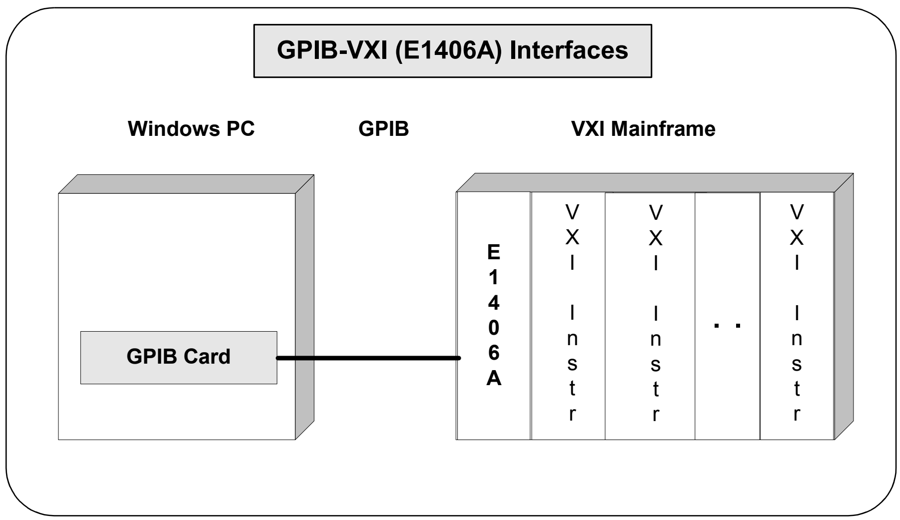
GPIB-VXI Interfaces Overview

As shown in the following figure, a typical GPIB-VXI interface consists of a GPIB card (82350 or equivalent) in a Windows PC that is connected via a GPIB cable to an E1406A Command Module. The E1406A sends commands to the VXI instruments in a VXI mainframe. There is no direct access to the VXI backplane from the PC.

.

NOTE

For a GPIB-VXI interface, VISA uses a DLL supplied by the Command Module vendor to translate the VISA VXI calls to Command Module commands that are vendor-specific. The DLL required for Agilent/Hewlett-Packard Command Modules is installed by the Agilent IO Libraries Installer. This DLL is installed by default when Agilent VISA is installed.



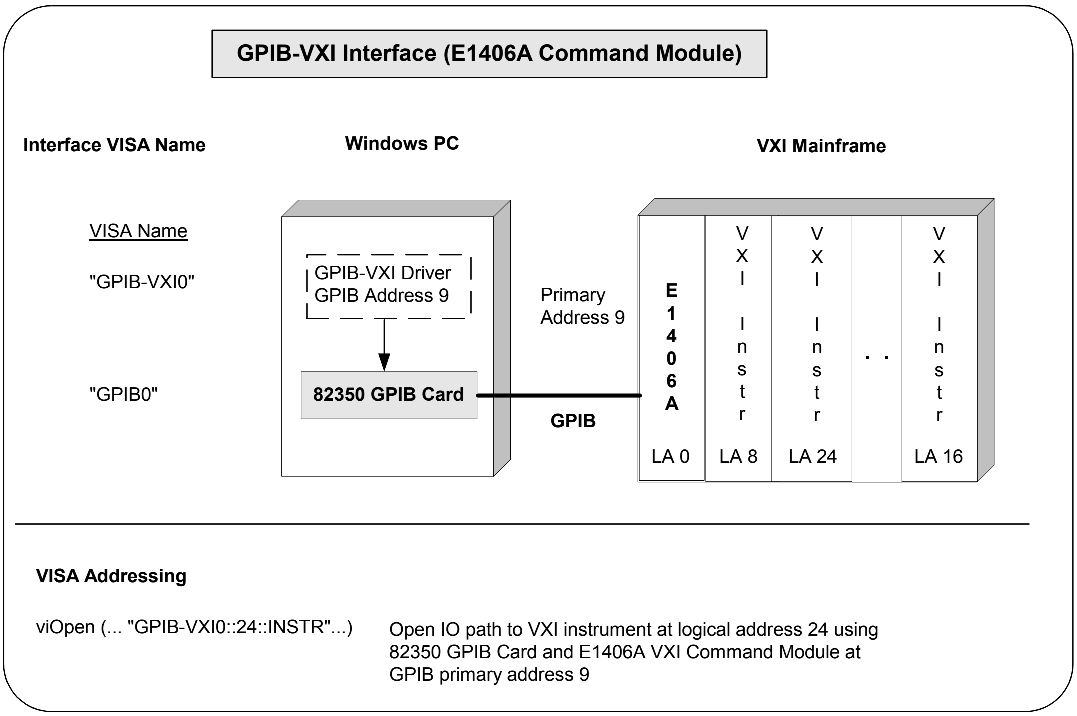
Example: GPIB-VXI (E1406A) Interface

The GPIB-VXI interface system in the following figure consists of a Windows PC with an 82350 GPIB card that connects to an E1406A Command Module in a VXI Mainframe. The VXI mainframe includes one or more VXI instruments.

When the IO Libraries were installed, a GPIB-VXI driver with GPIB address 9 was also installed and the E1406A was configured for primary address 9 and logical address (LA) 0. The three VXI instruments shown have logical addresses 8, 16, and 24.

The IO Config utility has been used to assign the GPIB-VXI driver a VISA Name of **GPIB-VXI0** and to assign the 82350 GPIB card a VISA name of **GPIB0**. VISA addressing is as shown in the figure.

For information on the E1406A Command Module, see the *Agilent E1406A Command Module User's Guide*. For information on VXI instruments, see the applicable *VXI Instrument User's Guide*.



Using High-Level Memory Functions

High-level memory functions allow you to access memory on the interface through simple function calls. There is no need to map memory to a window. Instead, when high-level memory functions are used, memory mapping and direct register access are automatically done.

The tradeoff, however, is speed. High-level memory functions are easier to use. However, since these functions encompass mapping of memory space and direct register access, the associated overhead slows program execution time. If speed is required, use the low-level memory functions discussed in “Using Low-Level Memory Functions” later in this chapter.

Programming the Registers

High-level memory functions include the **viIn** and **viOut** functions for transferring 8-, 16-, or 32-bit values, as well as the **viMoveIn** and **viMoveOut** functions for transferring 8-, 16-, or 32-bit blocks of data into or out of local memory. You can therefore program using 8-, 16-, or 32-bit transfers.

High-Level Memory Functions

This table summarizes the high-level memory functions.

Table 25 Summary of High-Level Memory Functions

Function	Description
viIn8 (*vi, space, offset, val8* **) ;**	Reads 8 bits of data from the specified offset.
viIn16 (*vi, space, offset, val16* **) ;**	Reads 16 bits of data from the specified offset.
viIn32 (*vi, space, offset, val32* **) ;**	Reads 32 bits of data from the specified offset.
viOut8 (*vi, space, offset, val8* **) ;**	Writes 8 bits of data to the specified offset.
viOut16 (*vi, space, offset, val16* **) ;**	Writes 16 bits of data to the specified offset.

Table 25 Summary of High-Level Memory Functions

viOut32 (*vi, space, offset, val32* **) ;**	Writes 32 bits of data to the specified offset.
viMoveIn8(*vi, space, offset, length, buf8* **) ;**	Moves an 8-bit block of data from the specified offset to local memory.
viMoveIn16 (*vi, space, offset, length, buf16* **) ;**	Moves a 16-bit block of data from the specified offset to local memory.
viMoveIn32(*vi, space, offset, length, buf32* **) ;**	Moves a 32-bit block of data from the specified offset to local memory.
viMoveOut8(*vi, space, offset, length, buf8* **) ;**	Moves an 8-bit block of data from local memory to the specified offset.
viMoveOut16 (*vi, space, offset, length, buf16* **) ;**	Moves a 16-bit block of data from local memory to the specified offset.
viMoveOut32 (*vi, space, offset, length, buf32* **) ;**	Moves a 32-bit block of data from local memory to the specified offset.

Using *viIn* and *viOut*

When using the **viIn** and **viOut** high-level memory functions to program to the device registers, all you need to specify is the session identifier, address space, and the offset of the register. Memory mapping is done for you. For example, in this function:

```
viIn32(vi, space, offset, val32);
```

vi is the session identifier and *offset* is used to indicate the offset of the memory to be mapped. *offset* is relative to the location of this device's memory in the given address space. The *space* parameter determines which memory location to map the space. Valid *space* values are:

- **VI_A16_SPACE -** Maps in VXI/MXI A16 address space
- **VI_A24_SPACE -** Maps in VXI/MXI A24 address space
- **VI_A32_SPACE -** Maps in VXI/MXI A32 address space

The *val32* parameter is a pointer to where the data read will be stored. If instead you write to the registers via the **viOut32** function, the *val32* parameter is a pointer to the data to write to the specified registers. If the device specified by *vi* does not have memory in the specified address space, an error is returned. The following example uses **viIn16**.

```
ViSession defaultRM, vi;
ViUInt16 value;
.
viOpenDefaultRM(&&defaultRM);
viOpen(defaultRM, "VXI::24", VI_NULL, VI_NULL,
&vi);
viIn16(vi, VI_A16_SPACE, 0x100, &value);
```

Using *viMoveIn* and *viMoveOut*

You can also use the **viMoveIn** and **viMoveOut** high-level memory functions to move blocks of data to or from local memory. Specifically, the **viMoveIn** function moves an 8-, 16-, or 32-bit block of data from the specified offset to local memory, and the **viMoveOut** functions moves an 8-, 16-, or 32-bit block of data from local memory to the specified offset. Again, the memory mapping is done for you.

For example, in this function:

```
viMoveIn32(vi, space, offset, length, buf32);
```

vi is the session identifier and *offset* is used to indicate the offset of the memory to be mapped. *offset* is relative to the location of this device's memory in the given address space. The *space* parameter determines which memory location to map the space and the *length* parameter specifies the number of elements to transfer (8-, 16-, or 32-bits).

The *buf32* parameter is a pointer to where the data read will be stored. If instead you write to the registers via the **viMoveOut32** function, the *buf32* parameter is a pointer to the data to write to the specified registers.

High-Level Memory Functions Examples

Two example programs follow that use the high-level memory functions to read the ID and Device Type registers of a device at the VXI logical address 24. The contents of the registers are then printed out.

The first program uses the VXI interface and the second program accesses the backplane with the GPIB-VXI interface. These two programs are identical except for the string passed to **viOpen**.

Example: Using VXI Interface (High-Level) Memory Functions

This program uses high-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

```
/* vxihl.c
This example program uses the high-level memory
functions to read the id and device type
registers of the device at VXI0::24. Change this
address if necessary. The register contents are
then displayed.*/

#include <visa.h>
#include <stdlib.h>
#include <stdio.h>

void main () {

   ViSession defaultRM, dmm;
   unsigned short id_reg, devtype_reg;

  /* Open session to VXI device at address 24 */
   viOpenDefaultRM(&defaultRM);
   viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL,
   VI_NULL, &dmm);

   /* Read instrument id register contents */
   viIn16(dmm, VI_A16_SPACE, 0x00, &id_reg);
```

```
    /* Read device type register contents */
    viIn16(dmm, VI_A16_SPACE, 0x02,
    &devtype_reg);

    /* Print results */
    printf ("ID Register = 0x%4X\n", id_reg);
    printf ("Device Type Register = 0x%4X\n",
    devtype_reg);

    /* Close sessions */
    viClose(dmm);
    viClose(defaultRM);
}
```

Example: Using GPIB-VXI Interface (High-Level) Memory Functions

This program uses high-level memory functions and the GPIB-VXI interface to read the ID and Device Type registers of a device at GPIB-VXI0::24.

```
/*gpibvxih.c
This example program uses the high-level memory
functions to read the id and device type
registers of the device at GPIB-VXI0::24. Change
this address if necessary. The register
contents are then displayed.*/

#include <visa.h>
#include <stdlib.h>
#include <stdio.h>

void main ()
   {
   ViSession defaultRM, dmm;
   unsigned short id_reg, devtype_reg;

   /* Open session to VXI device at address 24 */
   viOpenDefaultRM(&defaultRM);
   viOpen(defaultRM, "GPIB-VXI0::24::INSTR",
   VI_NULL,VI_NULL, &dmm);
```

```
/* Read instrument id register contents */
viIn16(dmm, VI_A16_SPACE, 0x00, &id_reg);

/* Read device type register contents */
viIn16(dmm, VI_A16_SPACE, 0x02,
&devtype_reg);

/* Print results */
printf ("ID Register = 0x%4X\n", id_reg);
printf ("Device Type Register = 0x%4X\n",
devtype_reg);

/* Close sessions */
viClose(dmm);
viClose(defaultRM);
}
```

Using Low-Level Memory Functions

Low-level memory functions allow direct access to memory on the interface just as do high-level memory functions. However, with low-level memory function calls, you must map a range of addresses and directly access the registers with low-level memory functions, such as **viPeek32** and **viPoke32**.

There is more programming effort required when using low-level memory functions. However, the program execution speed can increase. Additionally, to increase program execution speed, the low-level memory functions do not return error codes.

Programming the Registers

When using the low-level memory functions for direct register access, you must first map a range of addresses using the **viMapAddress** function. Next, you can send a series of peeks and pokes using the **viPeek** and **viPoke** low-level memory functions. Then, you must free the address window using the **viUnmapAddress** function. A process you could use is:

1. Map memory space using **viMapAddress**.
2. Read and write to the register's contents using **viPeek32** and **viPoke32**.
3. Unmap the memory space using **viUnmapAddress**.

Low-Level Memory Functions

You can program the registers using low-level functions for 8-, 16-, or 32-bit transfers. This table summarizes the low-level memory functions.

Table 26 Summary of Low-Level Memory Functions

Function	Description
viMapAddress(*vi, mapSpace, mapBase, mapSize, access, suggested, address*);	Maps the specified memory space.

Table 26 Summary of Low-Level Memory Functions

viPeek8(*vi, addr, val8*);	Reads 8 bits of data from address specified.
viPeek16(*vi, addr, val16*);	Reads 16 bits of data from address specified.
viPeek32(*vi, addr, val32*);	Reads 32 bits of data from address specified.
viPoke8(*vi, addr, val8*);	Writes 8 bits of data to address specified.
viPoke16(*vi, addr, val16*);	Writes 16 bits of data to address specified.
viPoke32(*vi, addr, val32*);	Writes 32 bits of data to address specified.
viUnmapAddress(*vi*);	Unmaps memory space previously mapped.

Mapping Memory Space

When using VISA to access the device's registers, you must map memory space into your process space. For a given session, you can have only one map at a time. To map space into your process, use the VISA **viMapAddress** function:

```
viMapAddress(vi, mapSpace, mapBase, mapSize,
access, suggested, address);
```

This function maps space for the device specified by the *vi* session. *mapBase*, *mapSize*, and *suggested* are used to indicate the offset of the memory to be mapped, amount of memory to map, and a suggested starting location, respectively. *mapSpace* determines which memory location to map the space. The following are valid *mapSpace* choices:

VI_A16_SPACE - Maps in VXI/MXI A16 address space

VI_A24_SPACE - Maps in VXI/MXI A24 address space

VI_A32_SPACE - Maps in VXI/MXI A32 address space

A pointer to the address space where the memory was mapped is returned in the *address* parameter. If the device specified by *vi* does not have memory in the specified address space, an error is returned. Some example **viMapAddress** function calls follow.

```
/* Maps to A32 address space */
viMapAddress(vi, VI_A32_SPACE, 0x000, 0x100,
VI_FALSE,
   VI_NULL,&address);
```

```
/* Maps to A24 address space */
viMapAddress(vi, VI_A24_SPACE, 0x00, 0x80,
VI_FALSE,
   VI_NULL,&address);
```

Reading and Writing to Device Registers

When you have mapped the memory space, use the VISA low-level memory functions to access the device's registers. First, determine which device register you need to access. Then, you need to know the register's offset. See the applicable instrument's user manual for a description of the registers and register locations. You can then use this information and the VISA low-level functions to access the device registers.

Example: Using *viPeek16*

An example using **viPeek16** follows.

```
ViSession defaultRM, vi;
ViUInt16 value;
ViAddr address;
ViUInt16 value;
.
.
viOpenDefaultRM(&&defaultRM);
viOpen(defaultRM, "VXI::24::INSTR", VI_NULL,
VI_NULL,
   &vi);
viMapAddress(vi, VI_A16_SPACE, 0x00, 0x04,
VI_FALSE,
   VI_NULL, &address);
viPeek16(vi, addr, &value)
```

Unmapping Memory Space

Make sure you use the **viUnmapAddress** function to unmap the memory space when it is no longer needed. Unmapping memory space makes the window available for the system to reallocate.

Low-Level Memory Functions Examples

Two example programs follow that use the low-level memory functions to read the ID and Device Type registers of the device at VXI logical address 24. The contents of the registers are then printed out. The first program uses the VXI interface and the second program uses the GPIB-VXI interface to access the VXI backplane. These two programs are identical except for the string passed to **viOpen**.

Example: Using the VXI Interface (Low-Level) Memory Functions

This program uses low-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

```
/*vxill.c
This example program uses the low-level memory
functions to read the id and device type
registers of the device at VXI0::24. Change this
address if necessary. The register contents are
then displayed.*/

#include <visa.h>
#include <stdlib.h>
#include <stdio.h>

void main () {

   ViSession defaultRM, dmm;
   ViAddr address;
   unsigned short id_reg, devtype_reg;

  /* Open session to VXI device at address 24 */
   viOpenDefaultRM(&defaultRM);
   viOpen(defaultRM, "VXI0::24::INSTR", VI_NULL,
      VI_NULL, &dmm);
```

```
/* Map into memory space */
viMapAddress(dmm, VI_A16_SPACE, 0x00, 0x10,
   VI_FALSE,VI_NULL, &address);

/* Read instrument id register contents */
viPeek16(dmm, address, &id_reg);

/* Read device type register contents */
/* ViAddr is defined as a void so we must cast
/* it to something else to do pointer
 arithmetic */
viPeek16(dmm, (ViAddr)((ViUInt16 *)address +
 0x01),
  &devtype_reg);

/* Unmap memory space */
viUnmapAddress(dmm);

/* Print results */
printf ("ID Register = 0x%4X\n", id_reg);
printf ("Device Type Register = 0x%4X\n",
 devtype_reg);

/* Close sessions */
viClose(dmm);
viClose(defaultRM);
}
```

Example: Using the GPIB-VXI Interface (Low-Level) Memory Functions

This program uses low-level memory functions and the GPIB-VXI interface to read the ID and Device Type registers of a device at GPIB-VXI0::24.

```
/*gpibvxil.c
This example program uses the low-level memory
functions to read the id and device type
registers of the device at GPIB-VXI0::24. Change
this address if necessary. Register contents are
then displayed.*/
```

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

void main () {

   ViSession defaultRM, dmm;
   ViAddr address;
   unsigned short id_reg, devtype_reg;

  /* Open session to VXI device at address 24 */
   viOpenDefaultRM(&defaultRM);
   viOpen(defaultRM, "GPIB-VXI0::24::INSTR",
    VI_NULL,
      VI_NULL,&dmm);

   /* Map into memory space */
   viMapAddress(dmm, VI_A16_SPACE, 0x00, 0x10,
    VI_FALSE,
      VI_NULL, &address);

   /* Read instrument id register contents */
   viPeek16(dmm, address, &id_reg);

   /* Read device type register contents */
   /* ViAddr is defined as a void so we must
    cast it to something else to do pointer
    arithmetic */
   viPeek16(dmm, (ViAddr)((ViUInt16 *)address +
    0x01),
      &devtype_reg);

   /* Unmap memory space */
   viUnmapAddress(dmm);

   /* Print results */
   printf ("ID Register = 0x%4X\n", id_reg);
   printf ("Device Type Register = 0x%4X\n",
    devtype_reg);

   /* Close sessions */
   viClose(dmm);
   viClose(defaultRM);
   }
```

Using Low/High-Level Memory I/O Methods

VISA supports three different memory I/O methods for accessing memory on the VXI backplane, as shown. All three of these access methods can be used to read and write VXI memory in the A16, A24, and A32 address spaces. The best method to use depends on the VISA program characteristics.

- Low-level **viPeek/viPoke**
  - viMapAddress
  - viUnmapAddress
  - **viPeek8, viPeek16, viPeek32**
  - **viPoke8, viPoke16, viPoke32**

- High-level **viIn/viOut**
  - **viIn8, viIn16, viIn32**
  - **viOut8, viOut16, viOut32**

- High-level **viMoveIn/viMoveOut**
  - viMoveIn8, viMoveIn16, viMoveIn32
  - **viMoveOut8, viMoveOut16, viMoveOut32**

Using Low-Level viPeek/viPoke

Low-level **viPeek/viPoke** is the most efficient in programs that require repeated access to different addresses in the same memory space.

The advantages of low-level **viPeek/viPoke** are:

- Individual **viPeek/viPoke** calls are faster than **viIn/viOut** or **viMoveIn/viMoveOut** calls.
- Memory pointer may be directly de-referenced in some cases for the lowest possible overhead.

The disadvantages of low-level **viPeek/viPoke** are:

- **viMapAddress** call is required to set up mapping before **viPeek/viPoke** can be used.

- **viPeek/viPoke** calls do not return status codes.
- Only one active **viMapAddress** is allowed per *vi* session.
- There may be a limit to the number of simultaneous active **viMapAddress** calls per process or system.

Using High-level viIn/viOut

High-level **viIn/viOut** calls are best in situations where a few widely scattered memory accesses are required and speed is not a major consideration.

The advantages high-level **viIn/viOut** are:

- Simplest method to implement.
- No limit on number of active maps.
- A16, A24, and A32 memory access can be mixed in a single *vi* session.

The disadvantage of high-level **viIn/viOut** calls is that they are slower than **viPeek/viPoke**.

Using High-level viMoveIn/viMoveOut

High-level **viMoveIn/viMoveOut** calls provide the highest possible performance for transferring blocks of data to or from the VXI backplane. Although these calls have higher initial overhead than the **viPeek/viPoke** calls, they are optimized on each platform to provide the fastest possible transfer rate for large blocks of data.

For small blocks, the overhead associated with **viMoveIn/voMoveOut** may actually make these calls longer than an equivalent loop of **viIn/viOut** calls. The block size at which **viMoveIn/viMoveOut** becomes faster depends on the particular platform and processor speed.

The advantages of high-level **viMoveIn/viMoveOut** are:

- Simple to use.
- No limit on number of active maps.
- A16, A24, and A32 memory access can be mixed in a single *vi* session.

- Provides the best performance when transferring large blocks of data.
- Supports both block and FIFO mode.

The disadvantage of **viMoveIn/viMoveOut** calls is that they have higher initial overhead than **viPeek/viPoke**.

Example: Using VXI Memory I/O

This program demonstrates using various types of VXI memory I/O.

```
/* memio.c
This example program demonstrates the use of
various memory I/O methods in VISA. */

#include <visa.h>
#include <stdlib.h>
#include <stdio.h>

#define VXI_INST "VXI0::24::INSTR"

void main () {
   ViSession defaultRM, vi;
   ViAddr         address;
   ViUInt16       accessMode;
   unsigned short *memPtr16;
   unsigned short id_reg;
   unsigned short devtype_reg;
   unsigned short memArray[2];

   /*Open default resource manager and session
   to instr*/
   viOpenDefaultRM (&defaultRM);
   viOpen defaultRM, VXI_INST, VI_NULL,VI_NULL,
   &vi);

/*
================================================
==
   Low level memory I/O = viPeek16 = direct
   memory dereference (when allowed)

================================================
```

```
*/
   /* Map into memory space */
   viMapAddress (vi, VI_A16_SPACE, 0x00, 0x10,
VI_FALSE,VI_NULL, &address);

/*
================================================
=
   Using viPeek

================================================
*/
   Read instrument id register contents */
   viPeek16 (vi, address, &id_reg);

   /* Read device type register contents
   ViAddr is defined as a (void *) so we must
   cast it to something else in order to do
   pointer arithmetic. */

   viPeek16 (vi, (ViAddr)((ViUInt16 *)address +
   0x01),&devtype_reg);

   /* Print results */
   printf ("   viPeek16: ID Register = 0x%4X\n",
   id_reg);
   printf ("   viPeek16: Device Type Register =
   0x%4X\n",devtype_reg);

   /* Use direct memory dereferencing if
   supported */
   viGetAttribute( vi, VI_ATTR_WIN_ACCESS,
   &accessMode );
   if ( accessMode == VI_DEREF_ADDR ) {

      /* assign pointer to variable of correct
      type */
      memPtr16 = (unsigned short *)address;

      /* do the actual memory reads */
      id_reg =      *memPtr16;
      devtype_reg = *(memPtr16+1);
```

```
       /* Print results */
       printf ("dereference: ID Register =
       0x%4X\n",id_reg);
      printf ("dereference: Device Type Register
      =0x%4X\n", devtype_reg);
       }

    /* Unmap memory space */
    viUnmapAddress (vi);


/*================================================
    High Level memory I/O = viIn16

================================================*/
    /* Read instrument id register contents */
    viIn16 (vi, VI_A16_SPACE, 0x00, &&id_reg);

    /* Read device type register contents */
    viIn16 (vi, VI_A16_SPACE, 0x02,&devtype_reg);

    /* Print results */
    printf ("  viIn16: ID Register = 0x%4X\n",
    id_reg);
    printf ("  viIn16: Device Type Register =
    0x%4X\n", devtype_reg);


/*================================================
======
High Level block memory I/O = viMoveIn16

The viMoveIn/viMoveOut commands do both block
read/write and FIFO read write. These commands
offer the best performance for reading and
writing large data blocks on the VXI backplane.
For this example we are only moving 2 words at a
time. Normally, these functions would be used to
move much larger blocks of data.

If the value of VI_ATTR_SRC_INCREMENT is 1 (the
default),viMoveIn does a block read. If the
value of VI_ATTR_SRC_INCREMENT is 0, viMoveIn
does a FIFO read.
```

```
If the value of VI_ATTR_DEST_INCREMENT is 1 (the
default),viMoveOut does a block write. If the
value of VI_ATTR_DEST_INCREMENT is 0, viMoveOut
does a FIFO write.
=============================================*/

/* Demonstrate block read.
   Read instrument id register and device type
   register into an array.*/

   viMoveIn16 (vi, VI_A16_SPACE, 0x00, 2,
   memArray);

   /* Print results */
   printf (" viMoveIn16: ID Register = 0x%4X\n",
      memArray[0]);
   printf (" viMoveIn16: Device Type Register =
      0x%4X\n", memArray[1]);

/* Demonstrate FIFO read.

   First set the source increment to 0 so we will
   repetitively read from the same memory
   location.*/
   viSetAttribute( vi, VI_ATTR_SRC_INCREMENT, 0
   );

   /* Do a FIFO read of the Id Register */
   viMoveIn16 (vi, VI_A16_SPACE, 0x00, 2,
   memArray);

   /* Print results */
   printf (" viMoveIn16: 1 ID Register =
   0x%4X\n",
      memArray[0]);
   printf (" viMoveIn16: 2 ID Register =
   0x%4X\n",
      memArray[1]);

   /* Close sessions */
   viClose (vi);
   viClose (defaultRM); }
```

Using the Memory Access Resource

For VISA 1.1 and later, the Memory Access (MEMACC) Resource type has been added to VXI and GPIB-VXI. VXI::MEMACC and GPIB-VXI::MEMACC allow access to all of the A16, A24, and A32 memory by providing the controller with access to arbitrary registers or memory addresses on memory-mapped buses.

The MEMACC resource, like any other resource, starts with the basic operations and attributes of other VISA resources. For example, modifying the state of an attribute is done via the the operation **viSetAttribute** (see *VISA Resource Classes* in the *VISA Online Help* for details).

Memory I/O Services

Memory I/O services include high-level memory I/O services and low-level memory I/O services.

High-Level Memory I/O Services

High-level Memory I/O services allow register-level access to the interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or even VME or VXI memory through a system controlled by a GPIB-VXI controller. A resource exists for each interface to which the controller has access.

You can access memory on the interface bus through operations such as **viIn16** and **viOut16**. These operations encapsulate the map/unmap and peek/poke operations found in the low-level service. There is no need to explicitly map the memory to a window.

Low-Level Memory I/O Services

Low-level Memory I/O services also allow register-level access to the interfaces that support direct memory access. Before an application can use the low-level service on the interface bus, it must map a range of addresses using the operation **viMapAddress**.

Although the resource handles the allocation and operation of the window, the programmer must free the window via **viUnMapAddress** when finished. This makes the window available for the system to reallocate.

Example: MEMACC Resource Program

This program demonstrates one way to use the MEMACC resource to open the entire VXI A16 memory and then calculate an offset to address a specific device.

```
/* peek16.c */
#include <stdio.h>
#include <stdlib.h>
#include <visa.h>

#define EXIT1
#define NO_EXIT 0

/* This function simplifies checking for VISA
errors. */
void checkError( ViSession vi, ViStatus status,
char
   *errStr,int doexit){
   char buf[256];
   if (status >= VI_SUCCESS)
      return;
   buf[0] = 0;
   viStatusDesc( vi, status, buf );
   printf( "ERROR 0x%lx (%s)\n '%s'\n", status,
errStr,
      buf );
   if ( doexit == EXIT )
      exit ( 1 );
   }

void main()  {
   ViSession drm;
   ViSession vi;
   ViUInt16inData16 = 0;
   ViUInt16peekData16 = 0;
   ViUInt8*addr;
```

```
   ViUInt16*addr16;
   ViStatusstatus;
   ViUInt16offset;

   status = viOpenDefaultRM ( &drm );
   checkError( 0, status, "viOpenDefaultRM",
EXIT );


   /* Open a session to the VXI MEMACC Resource*/
   status = viOpen( drm, "vxi0::memacc",
VI_NULL, VI_NULL,
      &vi );
   checkError (0, status, "viOpen", EXIT );

   /* Calculate the A16 offset of the VXI
REgisters for
      the device at VXI logical address 8. */
   offset = 0xc000 + 64 * 8;

   /* Open a map to all of A16 memory space. */
   status =
viMapAddress(vi,VI_A16_SPACE,0,0x10000,
      VI_FALSE,0,(ViPAddr)(&addr));
   checkError( vi, status, "viMapAddress", EXIT
);

   /* Offset the address pointer returned from
      viMapAddress for use with viPeek16. */
   addr16 = (ViUInt16 *) (addr + offset);

   /* Peek the contents of the card's ID register
(offset
      0 from card's base address. Note that
viPeek does
      not return a status code. */
   viPeek16( vi, addr16, &peekData16 );

   /* Now use viIn16 and read the contents of the
same
      register */
   status = viIn16(vi, VI_A16_SPACE,
      ViBusAddress)offset, &inData16 );
   checkError(vi, status, "viIn16", NO_EXIT );
```

```
   /* Print the results. */
   printf( "inData16 : 0x%04hx\n", inData16 );
   printf( "peekData16: ox%04hx\n", peekData16
);

   viClose( vi );
   viClose (drm );
   }
```

MEMACC Attribute Descriptions

Generic MEMACC Attributes

The following Read Only attributes (VI_ATTR_TMO_VALUE is Read/Write) provide general interface information

Table 27 Attributes That Provide General Interface Information

Attribute	Description
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of VI_TMO_IMMEDIATE means operation should never wait for the device to respond. A timeout value of VI_TMO_INFINITE disables the timeout mechanism.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_DMA_ALLOW_EN	Specifies whether I/O accesses should use DMA (VI_TRUE) or Programmed I/O (VI_FALSE).

VXI and GPIB-VXI Specific MEMACC Attributes

The following attributes, most of which are read/write, provide memory window control information.

Table 28 Attributes That Provide Memory Window Control Information

Attribute	Description
VI_ATTR_VXI_LA	Logical address of the local controller.
VI_ATTR_SRC_INCREMENT	Used in **viMoveInxx** operation to specify how much the source offset is to be incremented after every transfer. The default value is 1 and the **viMoveInxx** operation moves from consecutive elements. If this attribute is set to 0, the **viMoveInxx** operation will always read from the same element, essentially treating the source as a FIFO register.
VI_ATTR_DEST_INCREMENT	Used in **viMoveOutxx** operation to specify how much the destination offset is to be incremented after every transfer. The default value is 1 and the **viMoveOutxx** operation moves into consecutive elements. If this attribute is set to 0, the **viMoveOutxx** operation will always write to the same element, essentially treating the destination as a FIFO register.
VI_ATTR_WIN_ACCESS	Specifies modes in which the current window may be addressed: not currently mapped, through the **viPeekxx** or **viPokexx** operations only, or through operations and/or by directly de-referencing the address parameter as a pointer.
VI_ATTR_WIN_BASE_ADDR	Base address of the interface bus to which this window is mapped.
VI_ATTR_WIN_SIZE	Size of the region mapped to this window.

Table 28 Attributes That Provide Memory Window Control Information

VI_ATTR_SRC_BYTE_ORDER	Specifies the byte order used in high-level access operations, such as **viInxx** and **viMoveInxx**, when reading from the source.
VI_ATTR_DEST_BYTE_ORDER	Specifies the byte order used in high level access operations, such as **viOutxx** and **viMoveOutxx**, when writing to the destination.
VI_ATTR_WIN_BYTE_ORDER	Specifies the byte order used in low-level access operations, such as **viMapAddress**, **viPeekxx**, and **viPokexx**, when accessing the mapped window.
VI_ATTR_SRC_ACCESS_PRIV	Specifies the address modifier used in high-level access operations, such as **viInxx** and **viMoveInxx**, when reading from the source.
VI_ATTR_DEST_ACCESS_PRIV	Specifies the address modifier used in high-level access operations such as **viOutxx** and **viMoveOutxx**, when writing to destination.
VI_ATTR_WIN_ACCESS_PRIV	Specifies the address modifier used in low-level access operations, such as **viMapAddress**, **viPeekxx**, and **viPokexx**, when accessing the mapped window.

GPIB-VXI Specific MEMACC Attributes

The following Read Only attributes provide specific address information about GPIB hardware.

Table 29 Attributes that Provide Specific Address Information

Attribute	Description
VI_ATTR_INTF_PARENT_NUM	Board number of the GPIB board to which the GPIB-VXI is attached.
VI_ATTR_GPIB_PRIMARY_ADDR	Primary address of the GPIB-VXI controller used by the session.

Table 29 Attributes that Provide Specific Address Information

VI_ATTR_GPIB_SECONDARY_ADDR	Secondary address of the GPIB-VXI controller used by the session.

MEMACC Resource Event Attribute

The following Read Only events provide notification that an asynchronous operation has completed

Table 30 Events Providing Notification About Asynchronous Operations

Attribute	Description
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.
VI_ATTR_JOB_ID	Job ID of the asynchronous I/O operation that has completed.
VI_ATTR_BUFFER	Address of a buffer used in an asynchronous operation.
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.

Using VXI-Specific Attributes

VXI specific attributes can be useful to determine the state of your VXI system. Attributes are read only and read/write. Read only attributes specify things such as the logical address of the VXI device and information about where your VXI device is mapped. This section shows how you might use some of the VXI-specific attributes. See *VISA Resource Classes* in the *VISA Online Help* for information on VISA attributes.

Using the Map Address as a Pointer

The VI_ATTR_WIN_ACCESS read-only attribute specifies how a window can be accessed. You can access a mapped window with the VISA low-level memory functions or with a C pointer if the address is de-referenced. To determine how to access the window, read the VI_ATTR_WIN_ACCESS attribute.

VI_ATTR_WIN_ACCESS Settings

The VI_ATTR_WIN_ACCESS read-only attribute can be set to one of the following:

Table 31 Settings for the VI_ATTR_WIN_ACCESS Attribute

Setting	Description
VI_NMAPPED	Specifies that the window is not mapped.
VI_USE_OPERS	Specifies that the window is mapped and you can only use the low-level memory functions to access the data.
VI_DEREF_ADDR	Specifies that the window is mapped and has a de-referenced address. In this case you can use the low-level memory functions to access the data, or you can use a C pointer. Using a de-referenced C pointer will allow faster access to data.

Example: Determining Window Mapping

```
ViAddr address;
Vi UInt16 access;
ViUInt16 value;
.
.
.
viMapAddress(vi, VI_A16_SPACE, 0x00, 0x04,
VI_FALSE,
   VI_NULL, &address);
viGetAttribute(vi, VI_ATTR_WIN_ACCESS, &access);
.
.
If(access==VI_USE_OPERS) {
   viPeek16(vi, (ViAddr)(((ViUInt16 *)address) +
      4/sizeof(ViUInt16)), &value)
}else if (access==VI_DEREF_ADDR){
   value=*((ViUInt16
*)address+4/sizeof(ViUInt16));
}else if (access==VI_NMAPPED){
   return error;
}
.
.
```

Setting the VXI Trigger Line

The VI_ATTR_TRIG_ID attribute is used to set the VXI trigger line. This attribute is listed under generic attributes and defaults to VI_TRIG_SW (software trigger). To set one of the VXI trigger lines, set the VI_ATTR_TRIG_ID attribute as follows:

```
viSetAttribute(vi, VI_ATTR_TRIG_ID,
VI_TRIG_TTL0);
```

The above function sets the VXI trigger line to TTL trigger line 0 (VI_TRIG_TTL0). The following are valid VXI trigger lines. (Panel In is an Agilent extension of the *VISA specification.*)

Table 32 VXI Trigger Lines and Values

VXI Trigger Line	VI_ATTR_TRIG_ID Value
TTL 0	VI_TRIG_TTL0
TTL 1	VI_TRIG_TTL1
TTL 2	VI_TRIG_TTL2
TTL 3	VI_TRIG_TTL3
TTL 4	VI_TRIG_TTL4
TTL 5	VI_TRIG_TTL5
TTL 6	VI_TRIG_TTL6
TTL 7	VI_TRIG_TTL7
ECL 0	VI_TRIG_ECL0
ECL 1	VI_TRIG_ECL1
Panel In	VI_TRIG_PANEL_IN

Once you set a VXI trigger line, you can set up an event handler to be called when the trigger line fires. See Chapter 3, *"Programming with VISA"* for more information on setting up an event handler. Once the VI_EVENT_TRIG event is enabled, the VI_ATTR_TRIG_ID becomes a read only attribute and cannot be changed. You must set this attribute prior to enabling event triggers.

The VI_ATTR_TRIG_ID attribute can also be used by **viAssertTrigger** function to assert software or hardware triggers. If VI_ATTR_TRIG_ID is VI_TRIG_SW, the device is sent a Word Serial Trigger command. If the attribute is any other value, a hardware trigger is sent on the line corresponding to the value of that attribute.

5
Programming via LAN

This chapter provides guidelines for programming via a LAN (Local Area Network). A LAN is a way to extend the control of instrumentation beyond the limits of typical instrument interfaces.

The chapter contents are:

- LAN Interfaces Overview
- Communicating with LAN-Connected Devices

NOTE

This chapter describes programming using the VISA TCPIP interface type to communicate directly with a LAN-conected device, as well as using a VISA LAN client to emulate a GPIB, ASRL, or USB interface on the local machine to communicate with a LAN-connected device.

See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for LAN installation information and to start or stop the LAN servers.

LAN Interfaces Overview

This section provides an overview of LAN (Local Area Network) interfaces. A LAN is a way to extend the control of instrumentation beyond the limits of typical instrument interfaces. To communicate over the LAN, you must first configure the LAN Client interface. There are three main types of LAN interfaces:

- LAN Client
- VISA LAN Client
- LAN Server

LAN Hardware Architecture

The LAN software provided with the Agilent IO Libraries allows instrumentation control over a LAN. Using standard LAN connections, instruments can be controlled from computers that do not have special interfaces for instrument control.

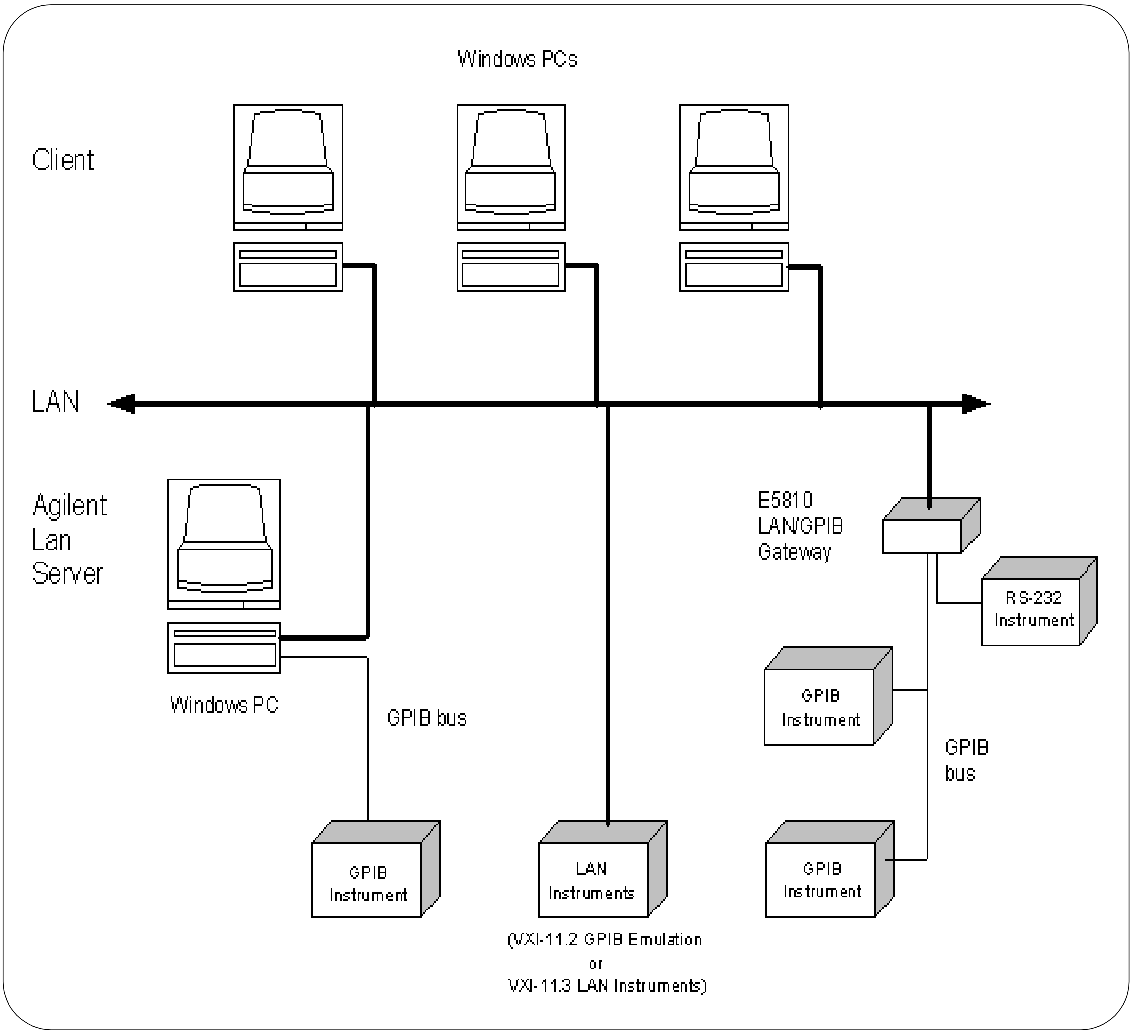
Client/Server Model

The LAN software uses the **client/server** model of computing. Client/server computing refers to a model where an application (the **client**) does not perform all necessary tasks of the application itself. Instead, the client makes requests of another computing device (the **server**) for certain services.

As shown in the following figure, a LAN client (such as a Series 700 HP-UX workstation or a Windows 98SE/Me/2000/XP/NT PC) makes VISA requests over the network to a LAN server (such as a Series 700 HP-UX workstation, a Windows 98SE/Me/2000/XP/NT PC, or an E5810 LAN/GPIB Gateway).

Gateway Operation

The LAN server is connected to the instrumentation or devices to be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains requested data and status information that indicates whether or not the operation was successful. The LAN server acts as a **gateway** between the LAN software that the client system supports and the instrument-specific interface that the device supports.

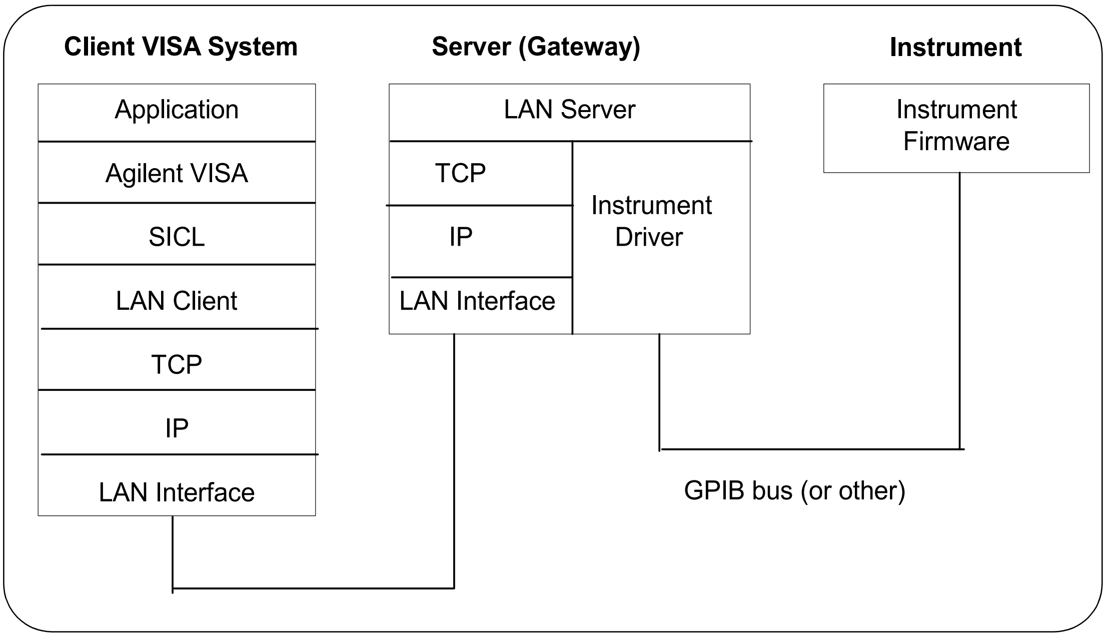


LAN Software Architecture

An **IO interface** can be defined as both a hardware interface and as a software interface. You can use the IO Config utility to associate a unique interface name with a hardware interface. The IO Libraries use a **VISA Interface Name** to identify an interface. This information is passed in the parameter string of the **viOpen** function call in a VISA program.

IO Config assigns a VISA Interface Name to the interface hardware, as well as other necessary configuration values for an interface when the interface is configured. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details on using IO Config.

As shown in the following figure, the client system contains the LAN client software and the LAN software (TCP/IP) needed to access the server (gateway). The gateway contains the LAN server software, LAN (TCP/IP) software, and the instrument driver software needed to communicate with the client and to control the instruments or devices connected to the gateway.



The LAN software is built on top of standard LAN networking protocols. There are two LAN networking protocols provided with the Agilent IO Libraries software. You can use one or both of these protocols when configuring your systems (via Agilent IO Libraries configuration) to use VISA over a LAN.

- **SICL-LAN Protocol** is a networking protocol developed by Agilent that is compatible with all VISA LAN products. This LAN networking protocol is the default choice in the Agilent IO Libraries configuration when configuring the LAN client. The SICL-LAN protocol on Windows 98SE/Me/2000/XP/NT supports VISA operations over LAN to GPIB interfaces.
- **VXI-11 (TCP/IP Instrument Protocol)** is a networking protocol developed by the VXIbus Consortium based on the SICL-LAN Protocol that permits interoperability of LAN software from different vendors who meet the VXIbus Consortium standards.

When using either of these networking protocols, the LAN software uses the TCP/IP protocol suite to pass messages between the LAN client and the LAN server. The server accepts device I/O requests over the network from the client and then proceeds to execute those I/O requests on a local interface (GPIB, etc.).

By default, the LAN Client supports both protocols by automatically detecting the protocol the server is using. When a VISA **viOpen** is performed, the LAN Client driver first tries to connect using the SICL-LAN protocol. If that fails, the driver will try to connect using the VXI-11 protocol.

If you want to control the protocol used, you can configure more than one LAN Client interface and set each interface to a different protocol. The protocol used will then depend on the interface you are connecting through.

Thus, you can have more than one SICL-LAN and one VXI-11 protocol for your system. In VISA, the protocol used is determined by the configuration settings and cannot be changed programatically. The LAN Client also supports TCP/IP socket reads and writes.

When you have configured VISA LAN Client interfaces, you can then use the interface name specified during configuration in a VISA **viOpen** call of your program. However, the LAN server does *not* support simultaneous connections from LAN clients using the SICL-LAN Protocol and from LAN clients using VXI-11 (TCP/IP Instrument Protocol).

There are three LAN servers that can be used with VISA: the E2050 LAN/GPIB Gateway, the E5810 LAN/GPIB Gateway, or a PC running Windows 98SE/Me/2000/XP/NT. To use this capability, the LAN server must have a local GPIB interface configured for I/O.

LAN Client Interface Overview

There are two main configurations for a LAN Client interface:

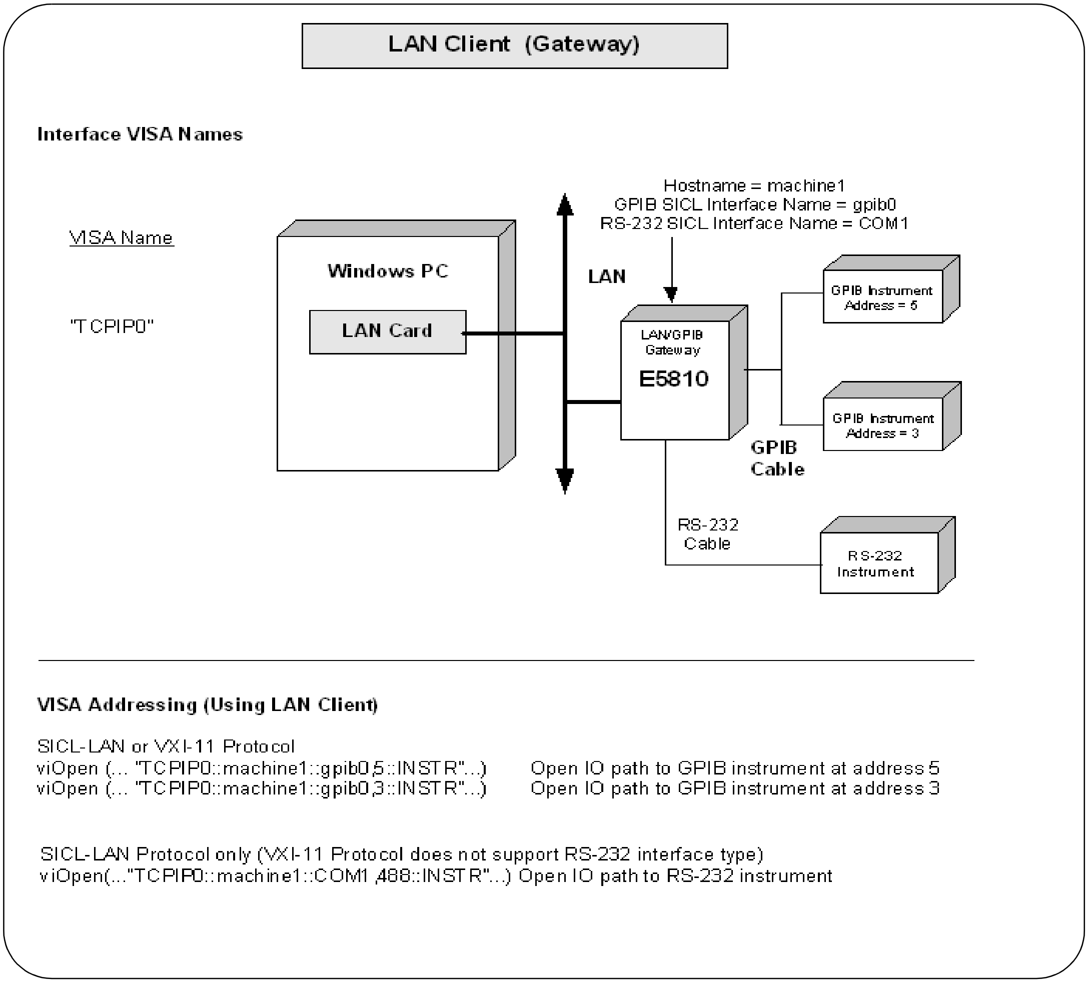
- LAN Client (Gateway)
- LAN Client (LAN)

This section provides an example of each configuration and shows applicable VISA **viOpen** commands. See the *VISA Online Help* for details on the VISA commands.

Example: LAN Client (Gateway) Interface

The LAN Client interface system in the following figure consists of a Windows PC with a LAN card, an E5810 LAN/GPIB gateway, and two GPIB instruments. For this system, the IO Config utility has been used to assign the LAN card a VISA name of **TCPIP0**.

With this name assigned to the interface, VISA addressing is as shown in the figure, and you can use the VISA **viOpen** command to open the I/O paths to the GPIB instruments as shown.

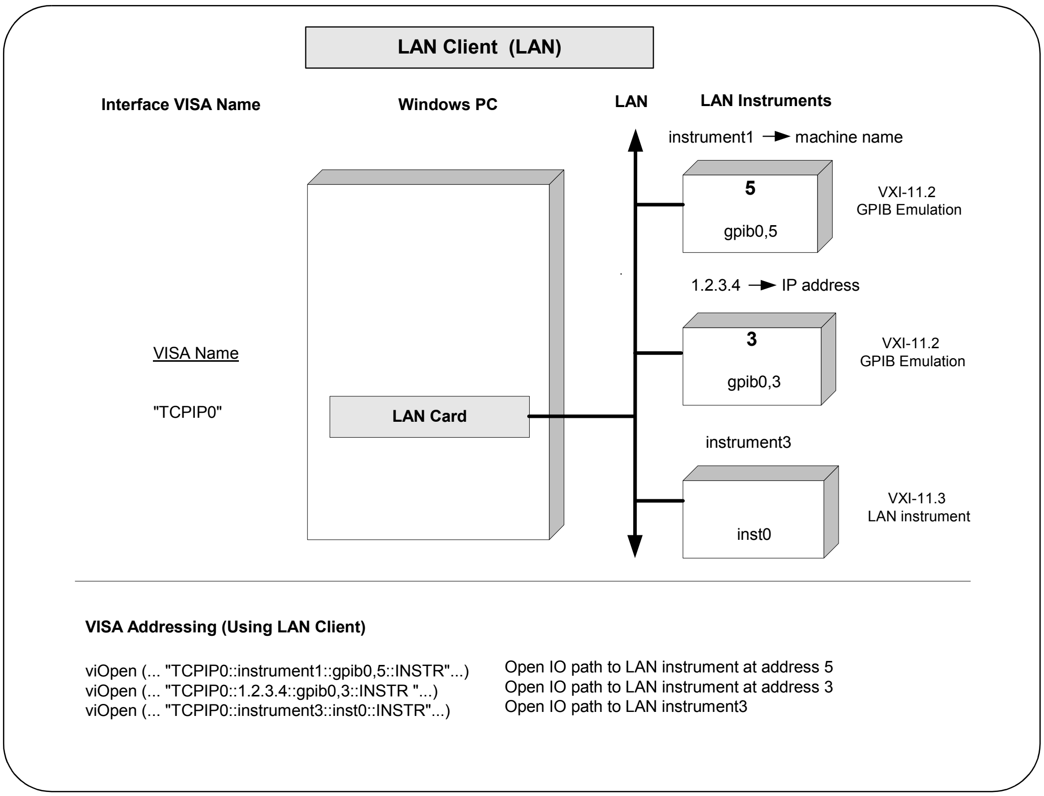


Example: LAN Client (LAN) Interface

The LAN Client interface system in the following figure consists of a Windows PC with a LAN card and three LAN instruments. Instrument1 and instrument2 are VXI-11.2 (GPIB Emulation) instruments and instrument3 is a VXI-11.3 LAN instrument.

For this system, the IO Config utility has been used to assign the LAN card a VISA name of **TCPIP0**. For the addressing examples, instrument1 has been addressed by its machine name, instrument 2 has been addressed by its IP address, and instrument3 by its LAN name (**inst0**).

Since unique names have been assigned by IO Config, you can now use the VISA **viOpen** command to open the I/O paths to the GPIB instruments as shown in the figure.



VISA LAN Client Interface Overview

There are two main configurations for a VISA LAN Client interface:

- VISA LAN Client (Gateway)
- VISA LAN Client (LAN)

This section provides an example of each configuration and shows applicable VISA **viOpen** commands. See the *VISA Online Help* for details on the VISA commands.

NOTE

A VISA LAN Client interface requires a LAN Client interface. When a VISA LAN Client interface is configured, it automatically configures a LAN Client interface if one is not already configured. See "Configuring LAN Client Interfaces" in the *Agilent IO Libraries Installation and Configuration Guide for Windows* for details on configuring a LAN Client interface.

Example: VISA LAN Client (Gateway) Interface

The VISA LAN Client interface system in the following figure consists of a Windows PC with a LAN card, an E5810 LAN/GPIB gateway, two GPIB instruments, and an RS-232 (ASRL) instrument. The IO Config utility has been used to assign the LAN card a VISA name of **TCPIP0**.

In addition, a GPIB VISA LAN Client and an ASRL VISA LAN client have been configured with the interface names and host names shown in the figure. The E5810 LAN/GPIB Gateway has been assigned a Hostname of **machine1**, a GPIB SICL Interface Name = **gpib0**, and an RS-232 Interface Name = **COM1**.

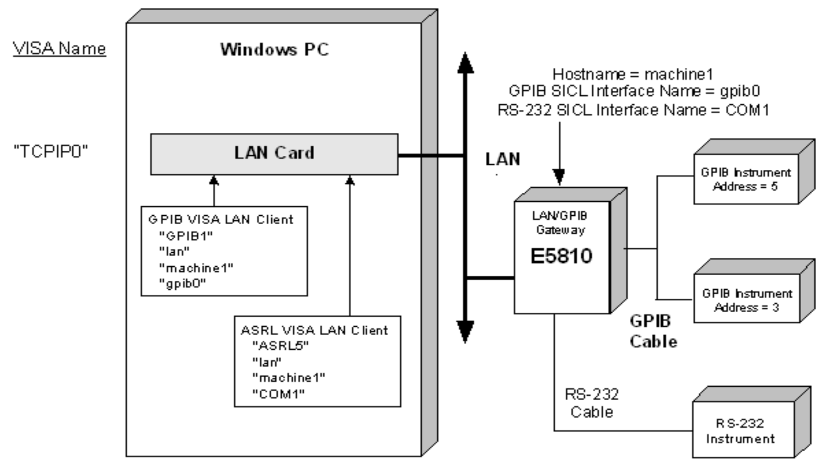
Since unique names have been assigned by IO Config, you can now use the VISA **viOpen** command to open the I/O paths to the GPIB and RS-232 instruments as shown in the figure.

NOTE

The SICL-LAN protocol supports both the GPIB VISA LAN Client and the ASRL VISA LAN Client. The VXI-11 protocol supports only the GPIB-VISA LAN Client.

VISA LAN Client (Gateway)

Interface VISA Names



VISA LAN Client Parameters

VISA Interface Name	"GPIB1"
LAN Client SICL Interface Name	"lan"
Remote Host Name	"machine1"
Remote SICL Interface Name	"gpib0"

ASRL VISA LAN Client Parameters *(Requires SICL-LAN protocol)*

VISA Interface Name	"ASRL5"
LAN Client SICL Interface Name	"lan"
Remote Host Name	"machine1"
Remote SICL Interface Name	"COM1"

VISA Addressing (Using LAN Client)

viOpen (... "TCPIP0::machine1::gpib0,5::INSTR"...)	Open IO path to remote GPIB instrument at address 5
viOpen (... "TCPIP0::machine1::gpib0,3::INSTR"...)	Open IO path to remote GPIB instrument at address 3
viOpen (... "TCPIP0::machine1::COM1::INSTR"...)	Open IO path to remote RS-232 instrument (requires SICL-LAN protocol)

VISA Addressing (Using VISA LAN Client)

viOpen (... "GPIB1::5::INSTR"...)
viOpen (... "GPIB1::3::INSTR "...)
viOpen (... "ASRL5::INSTR "...)

Example: VISA LAN Client (LAN) Interface

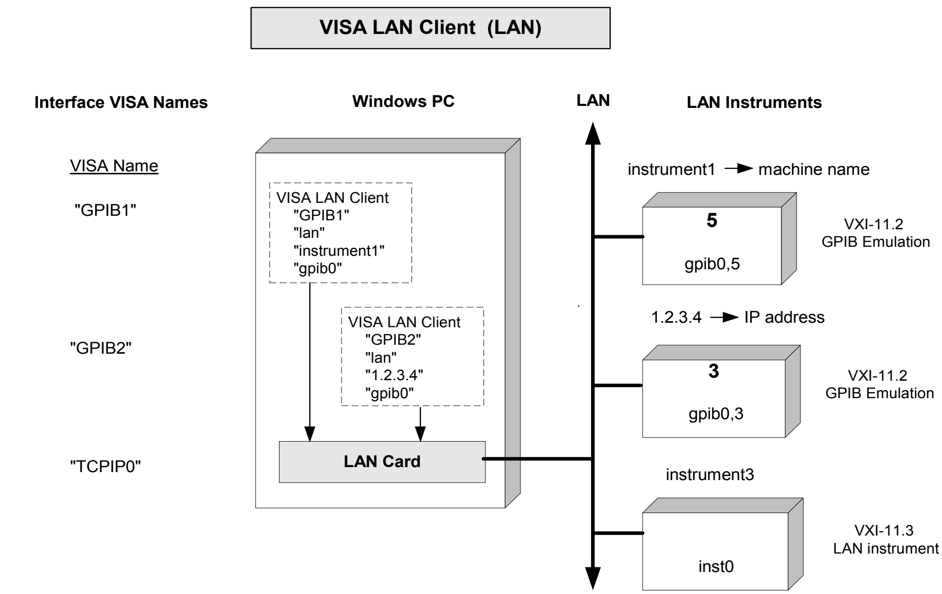
The VISA LAN Client interface system in the following figure consists of a Windows PC with a LAN card and three LAN instruments. Instrument1 and instrument2 are VXI-11.2 (GPIB Emulation) instruments and instrument3 is a VXI-11.3 LAN instrument.

For this system, the IO Config utility has been used to assign the LAN card a VISA name of **TCPIP0**. In addition, two GPIB VISA LAN Clients have been configured with the interface names and host names shown in the figure.

For the addressing examples, instrument1 has been addressed by its hostname, instrument2 has been addressed by its IP address, and instrument3 by its LAN name (**inst0**).

Since unique names have been assigned by IO Config, you can now use the VISA **viOpen** command to open the I/O paths to the GPIB instruments as shown in the figure. Note, however, that you cannot talk to instrument3 with a VISA LAN Client. You must use the LAN Client to talk to instrument3, since instrument3 is not a remote GPIB interface.

NOTE When using the VXI-11 protocol with GPIB VISA LAN Client, the Remote SICL Interface Name must be of the form **gpibN** where **N** is **0** or a positive integer. This restriction does not apply to the SICL-LAN protocol.



VISA LAN Client Parameters	GPIB1	GPIB2
VISA Interface Name	"GPIB1"	"GPIB2"
LAN Client SICL Interface Name	"lan"	"lan"
Remote Host Name	"instrument1"	"1.2.3.4"
Remote SICL Interface Name	"gpib0"	"gpib0"

VISA Addressing (Using LAN Client)

viOpen (... "TCPIP0::instrument1::gpib0,5::INSTR"...)	Open IO path to LAN instrument at address 5
viOpen (... "TCPIP0::1.2.3.4::gpib0,3::INSTR "...)	Open IO path to LAN instrument at address 3
viOpen (... "TCPIP0::instrument3::inst0::INSTR"...)	Open IO path to LAN instrument3

VISA Addressing (Using VISA LAN Client)

VISA: viOpen (... "GPIB1::5::INSTR"...)	Open IO path to LAN instrument at address 5
viOpen (... "GPIB2::3::INSTR "...)	Open IO path to LAN instrument at address 3
Cannot talk to instrument3	

LAN Server Interface Overview

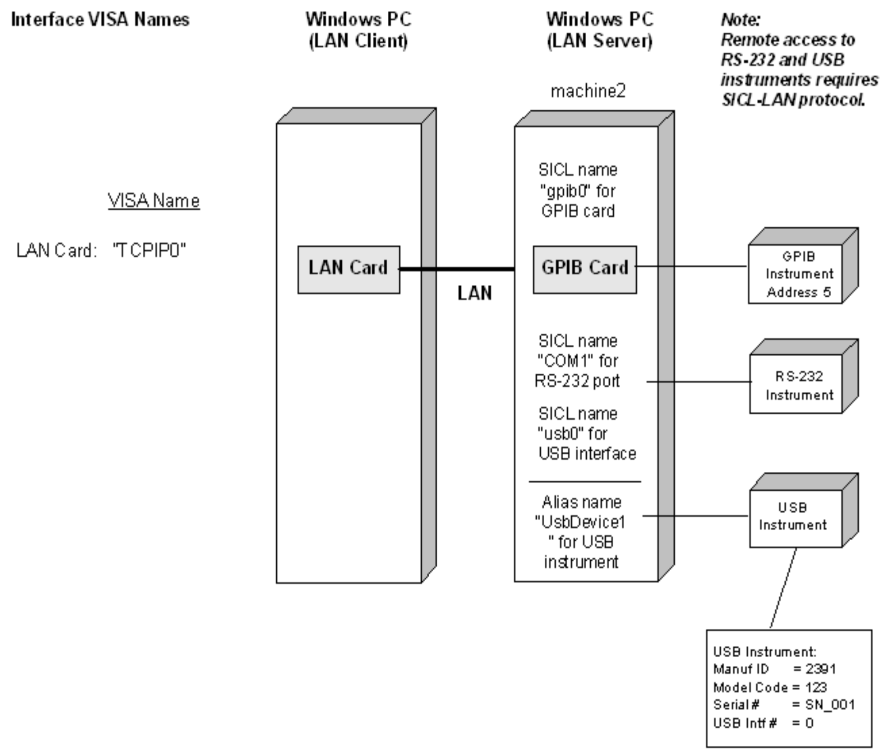
This section provides an example of the LAN Server interface configuration and shows applicable VISA **viOpen** commands. See the *VISA Online Help* for details on the VISA commands.

Example: LAN Server Interface

The LAN Server interface system in the following figure consists of a Windows PC acting as a LAN client, a second PC acting as a LAN server, with a GPIB instrument, an RS-232 (ASRL) instrument, and a USB instrument connected to it. The IO Config utility has been used to assign the LAN card a VISA name of **TCPIP0**. The LAN server PC has been assigned a hostname of **machine2**. Also, the GPIB card in the LAN server PC has been assigned the SICL name of **gpib0**, the RS-232 port has been assigned the SICL name **COM1**, and the USB instrument has been assigned the alias name **UsbDevice1**.

Since unique names have been assigned by IO Config, you can now use the VISA **viOpen** command to open the I/O paths to the GPIB instruments as shown in the figure.

LAN Server (PC as Server)



VISA Addressing

viOpen (... "TCPIP0::machine2::gpib0,5::INSTR".....)	Open IO path to GPIB instrument at address 5
viOpen (... "TCPIP0::machine2::COM1,488::INSTR".....)	Open IO path to RS-232 instrument
viOpen (... "TCPIP0::machine2::UsbDevice1::INSTR".....)	Open IO path to USB instrument with alias "UsbDevice1"

Alternate VISA Addressing for a USB device (without using an alias):

viOpen(..."TCPIP0::machine2::usb0[2391::123::SN_001::0]::INSTR"...)

Communicating with LAN-Connected Devices

VISA can communicate with LAN-connected devices in one of two ways:

- TCPIP interface type
- VISA LAN Client (available only with Agilent VISA)

Using the TCPIP Interface Type for LAN Access

VISA provides the TCPIP interface type to communicate with LAN-connected devices. These can be devices connected directly to the LAN, or they can be connected to the LAN through a LAN gateway such as the Agilent E5810 LAN/GPIB gateway or through a LAN server running on a remote computer with instruments connected to it.

The format of a TCPIP VISA resource string is:

```
TCPIP[<board>]::<hostname>[::<LAN device
name>][::INSTR]
```

where:

- `<board> =` board number (default is 0)
- `<hostname> =` the hostname or IP address of the LAN device or server
- `<LAN device name> =` the remote device name (case sensitive with default name of **inst0**)

The VXI-11 protocol constrains the LAN device name to be of the form **inst0**, **inst1**, ... for VXI-11.3 devices and **gpib0,n, gpib1,n**, ... for VXI-11.2 (GPIB Emulation) devices.

The SICL-LAN protocol allows any valid SICL name for the LAN device name. See the *Agilent SICL User's Guide for Windows* for additional information on valid SICL names.

Some examples of valid TCPIP resource strings that are valid for both the VXI-11 and SICL-LAN protocols are:

Table 33 Example TCPIP Resource Strings

String	Description
TCPIP0::testMachine@agilent.com::gpib0,2::INSTR	A VXI-11.2 GPIB device at hostname **testMachine@agilent.com**.
TCPIP0::123.456.0.21::gpib0,2::INSTR	A VXI-11.2 GPIB device at a machine whose IP Address is 123.456.0.21.
TCPIP0::myMachine::inst0::INSTR	A VXI-11.3 LAN instrument at hostname **myMachine**.
TCPIP::myMachine	A VXI-11.3 LAN instrument at hostname **myMachine**. Note that default values for **board = 0**, **LAN device name = inst0**, and the **::INSTR** resource class are used.
TCPIP0::testMachine1::COM1,488::INSTR	An RS-232 device connected to a LAN server or gateway at hostname **testMachine1**. This device must use SICL-LAN protocol since RS-232 devices are not supported by the VXI-11 protocol.
TCPIP0::myMachine::gpib0,2::INSTR	A GPIB device at hostname **myMachine**. This device must use SICL-LAN protocol since **gpib0,2** is not a valid remote name with the VXI-11 protocol.

Table 33 Example TCPIP Resource Strings

String	Description
TCPIP0::myMachine::UsbDevice1::INSTR	A USB device with a SICL alias of **UsbDevice1** connected to a LAN server at hostname **myMachine**. Note that the SICL alias is defined on the remote machine, not on the local machine. *Although the SICL and VISA alias names are usually the same, if they are not, you must be sure to use the SICL alias and not the VISA alias.* This device must use SICL-LAN protocol since USB devices are not supported by the VXI-11 protocol.
TCPIP0::myMachine::usb0[2391::1031::SN_00123::0]::INSTR	A USB device with: Manufacture ID = 2391 Model Code = 1031 Serial Number = 'SN_00123' USBTMC Intfc # = 0 connected to a LAN server at hostname **myMachine**. This device must use SICL-LAN protocol since USB devices are not supported by the VXI-11 protocol.

NOTE

A LAN session to a remote interface provides the same VISA function support as if the interface was local, except that VXI-specific functions are not supported over LAN.

Addressing a Session Using the TCPIP Interface Type

This example shows one way to open a device session with a GPIB device at primary address 23 on a remote PC that is running a LAN server. The hostname of the remote PC is **myMachine**. See Chapter 3, *"Programming with VISA"* for more information on addressing device sessions.

```
ViSession defaultRM, vi;.

.

viOpenDefaultRM(&defaultRM);

viOpen(defaultRM,
"TCPIP0::myMachine::gpib0,23::INSTR", VI_NULL,
VI_NULL, &vi);

.

.

viClose(vi);

viClose(defaultRM);
```

Using a VISA LAN Client for LAN Access

Agilent VISA provides three types of VISA LAN Client interfaces:

- ASRL VISA LAN Client
- GPIB VISA LAN Client
- USB VISA LAN Client

VISA LAN Client interfaces are configured using the Agilent IO Config utility and provide virtual GPIB, ASRL, or USB interfaces. This makes it possible to remotely access a LAN-connected device as if it were connected to a local interface. If, for example, the **GPIB2 VISA** interface is configured as a GPIB VISA LAN interface, a program controlling the devices **GPIB2::5::INSTR** and **GPIB2::7::INSTR** would not be aware of the fact that these devices are actually connected via LAN and not to a GPIB interface connected to the local machine.

See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for specific information on configuring VISA LAN Clients.

ASRL VISA LAN Client

An ASRL VISA LAN Client can use only the SICL-LAN protocol. An ASRL SICL LAN Client can be configured to use the serial port on the Agilent E5810 LAN/GPIB gateway or the serial ports on a PC running the LAN server.

GPIB VISA LAN Client

A GPIB VISA LAN Client can use both the VXI-11 and SICL-LAN protocols. Typical uses for GPIB VISA LAN Clients are with LAN/GPIB gateways (e.g. Agilent E5810), PCs with GPIB interfaces that are running a LAN server, and VXI-11.2 LAN-based instruments.

A GPIB VISA LAN Client can only be used to communicate with VXI-11.2 (GPIB Emulation) devices. This is because the VISA GPIB interface type requires a primary and (optionally) a secondary address when communicating with a device. VXI-11.3 devices do not support the concept of a primary address, so they cannot be accessed with a VISA LAN Client.

USB VISA LAN Client

A USB VISA LAN Client can use only the SICL-LAN protocol. It can communicate with USB devices attached to a remote PC running a LAN server.

Note that you must use the full USB resource string to access remote devices with the USB VISA LAN Client. Although an alias may have been assigned to the USB device on the remote system, the alias is not available on the local system.

Addressing a Session Using a VISA LAN Client

In general, the rules to address a LAN session are the same as to address a local session. The only difference for a LAN session is that you use the VISA Interface Name (provided during I/O configuration) that relates to the VISA LAN Client.

The following example shows one way to open a device session with a GPIB device at primary address 23 on a remote PC that is running a LAN server. A GPIB VISA LAN Client has been configured at GPIB2 to communicate with that machine. See Chapter 3, *"Programming with VISA"* for more information on addressing device sessions.

```
ViSession defaultRM, vi;.

.

viOpenDefaultRM(&defaultRM);

viOpen(defaultRM, "GPIB2::23::INSTR", VI_NULL,
VI_NULL, &vi);

.

.

viClose(vi);

viClose(defaultRM);
```

6
Programming via USB

This chapter provides guidelines for VISA programming of USB instruments that conform to USBTMC (Universal Serial Bus Test and Measurement Class) and/or USBTMC-USB488 (Universal Serial Bus Test and Measurement Class, Subclass USB488 Specification).

The chapter contents are:

USB Interfaces Overview

USBTMC/USBTMC-USB488 instruments are detected and automatically configured by Agilent VISA when they are plugged into the computer. The *Agilent IO Libraries Installation and Configuration Guide for Windows* describes the USB instrument configuration process in more detail.

NOTE

Do not confuse the Agilent 82357 USB/GPIB Interface with a USBTMC device. The 82357 is automatically configured as a GPIB interface, not as a USBTMC device, when it is plugged into the computer. Only USBTMC/USBTMC-USB488 devices are configured as USB devices by Agilent VISA.

Due to the complexity of the VISA USB resource string, an “alias” name is assigned to each USB instrument when it is plugged into the computer. You can use either the alias name or the full VISA resource string when opening a VISA resource, but using the alias name is recommended because it is simpler and allows substitution of USB instruments without the need to change a VISA program.

Communicating with a USB Instrument Using VISA

To establish communications with a USB device using VISA, you can use either the full VISA resource string for the device or use the alias name provided by VISA. Using the alias is recommended, for reasons described below.

Using the full VISA resource string, a **viOpen** call would look something like this:

```
viOpen( . . .,
"USB0::2391::1031::0000000123::0::INSTR", . . .
);
```

Following is a summary of the components of this call.

Table 34 Summary of Full-String viOpen Call

Value	Description	Data Type
2391	Manufacturer ID	16-bit unsigned integer
1031	Model Code	16-bit unsigned integer
0000000123	Serial Number	string value
0	USBTMC Interface Number	8-bit unsigned integer

This string uniquely identifies the USB device. The values needed for the resource string are displayed in a dialog box when the device is plugged into the computer.

To simplify the way a USB device is identified, Agilent VISA also provides an alias name which can be used in place of this resource string. The first USB device that is plugged in is assigned a default alias name of **UsbDevice1**. Additional devices are assigned aliases of **UsbDevice2**, **UsbDevice3**, etc. You can modify the default alias name at the time a device is plugged in, or by running the IO Configuration program and editing the USB interface.

Although the case of an alias name is preserved, case is ignored when the alias is used in place of the full resource string in an **iopen** call. For example, `UsbDevice1`, `usbdevice1` and `USBDEVICE1` all refer to the same device.

Using the alias name, a **viOpen** call would look something like this:

```
viOpen( . . ., "UsbDevice1", . . . );
```

As you can see, this is much simpler than having to use the full resource string for a USB device.

Using the alias name in a program also makes it more portable. For example, two identical USB function generators have different resource strings because they have different serial numbers. If these function generators are used in two different test systems and you use the full resource string to access the function generator in the test program, you cannot use that same program for both test systems, since the function generators' full resource strings are different. By using the alias name in the program, however, you can use the same program in both test systems. All you need to do is make sure the same alias name is used for the function generator in both systems.

Glossary

address

A string (or other language construct) that uniquely locates and identifies a resource. VISA defines an ASCII-based grammar that associates strings with particular physical devices or interfaces and VISA resources.

ADE

Application Development Environment.

API

Application Programmers Interface. The direct interface that an end user sees when creating an application. The VISA API consists of the sum of all of the operations, attributes, and events of each of the VISA Resource Classes.

attribute

A value within a resource that reflects a characteristic of the operational state of a resource. The operational state of some attributes can be changed.

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, non-existent memory, a non-existent register, or an incorrect device access can cause a bus error.

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

A device, such as a computer, used to communicate with a remote device, such as an instrument. In the communications between the controller and the device, the controller is in charge of and controls the flow of communication (that is, the controller does the addressing and/or other bus management).

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). In VISA, the concept of a device is generally the logical association of several VISA resources.

device session

A session that communicates as a controller specifically with a single device, such as an instrument.

handler

A software routine used to respond to an asynchronous event such as an SRQ or an interrupt.

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

Library of functions for controlling a specific instrument.

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

An asynchronous event requiring attention out of the normal flow of control of a program.

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.

operation

An action defined by a resource that can be performed on a resource.

process

An operating system component 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.

register

An address location that either contains a value that is a function of the state of hardware or can be written into to cause hardware to perform a particular action or to enter a particular state. In other words, an address location that controls and/or monitors hardware.

resource (or resource instance)

An instrument while using VISA. 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. In VISA, every defined software module is a resource.

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 that controls a particular capability of a device.

session

The same as Communication Channel. An instance of a communications path between a software element and a resource. Every communication channel in VISA is unique.

SRQ

IEEE-488 Service Request. This is an asynchronous request (an interrupt) from a remote GPIB 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 if the device is currently requesting service.

template function

Instrument driver subsystem function common to the majority of VXI*plug&play* instrument drivers.

thread

An operating system object that consists of a flow of control within a process. A single process may have multiple threads with each having access to the same data space within the process. However, each thread has its own stack and all threads may execute concurrently with each other (either on multiple processors, or by time-sharing a single processor). Note that multi-threaded applications are only supported with 32-bit VISA.

top-level example

A high-level test-oriented instrument driver function. It is typically developed from the instrument driver subsystem functions.

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. The VISA

Instrument Control Resources Organizer serves as a means to group any number of any type of VISA Instrument Control Resources within a VISA system.

VISA

Virtual Instrument Software Architecture. VISA is a common I/O library that allows software from different vendors to run together on the same platform. VISA is also the general name given to this document and its associated architecture. The architecture consists of two main VISA components: the VISA Resource Manager and the VISA Instrument Control 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 and interface capabilities for direct, low-level instrument control.

VISA resource manager

This is the name given to the part of VISA that manages resources. This management includes support for opening, closing, and finding resources, setting attributes, retrieving attributes, and generating events on resources, etc.

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.

Index

F

G

H

I

L

M

N

O

Q

R

S

T

U

V

W