Agilent VISA
User's Guide

Agilent Technologies

Manual Part Number: E2090-90040
Printed in U.S.A. E0701

Contents
Agilent VISA User's Guide

Notice

The information contained in this document is subject to change without notice.

Agilent Technologies shall not be liable for any errors contained in this document. *Agilent Technologies makes no warranties of any kind with regard to this document, whether express or implied. Agilent Technologies specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* Agilent Technologies shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Agilent Technologies product and replacement parts can be obtained from Agilent Technologies, Inc.

U.S. Government Restricted Rights

The Software and Documentation have been developed entirely at private expense. They are delivered and licensed as "commercial computer software" as defined in DFARS 252.227- 7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a "commercial item" as defined in FAR 2.101(a), or as "Restricted computer software" as defined in FAR 52.227-19 (Jun 1987) (or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and Documentation by the applicable FAR or DFARS clause or the Agilent standard software agreement for the product involved.

Trademark Information

Microsoft®, Windows ® 95, Windows ® 98, Windows ® Me, Windows ® 2000, and Windows NT® are U.S. registered trademarks of Microsoft Corporation. All other brand and product names are trademarks or registered trademarks of their respective companies.

Printing History

Edition 1 - May 1996
Edition 2 - September 1996
Edition 3 - February 1998
Edition 4 - July 2000
Edition 5 - July 2001

Copyright Information

1

Introduction

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 95, Windows 98, Windows Me, Windows NT 4.0, and Windows 2000, and on HP-UX version 10.20. This chapter includes:

- What's in This Guide?
- VISA Overview

NOTE

Before you can use VISA, you must install and configure VISA on your computer. See *Agilent IO Libraries Installation and Configuration Guide for Windows* for installation on Windows systems. See *Agilent IO Libraries Installation and Configuration for HP-UX* for installation on HP-UX 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 indicated, Windows NT refers to Windows NT 4.0.

What's in This Guide?

- *Chapter 1 - Introduction* describes the contents of this guide, provides an overview of VISA, and shows how to contact Agilent Technologies.
- *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 - Building a VISA Application in HP-UX* describes how to build a VISA application in the HP-UX environment. An example program is provided to help you get started programming with VISA.
- *Chapter 4 - 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 5 - Programming via GPIB and VXI* gives guidelines to use VISA to communicate over the GPIB, GPIB-VXI, and VXI interfaces to instruments.
- *Chapter 6 - Programming via LAN* gives guidelines to use VISA to communicate over a LAN (Local Area Network) to instruments.
- *Chapter 7 - VISA Language Reference* provides an alphabetical reference of supported VISA functions.
- *Appendix A - VISA Library Information* lists VISA data types and their definitions, VISA error codes, and VISA directory information.
- *Appendix B - VISA Resource Classes* describes the six VISA Resource Classes, including attributes, events, and operations.
- *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 and HP-UX environments.

For information on using SICL in Windows, see the *Agilent SICL User's Guide for Windows.* For information on using SICL in HP-UX, see the *Agilent Standard Instrument Control Library User's Guide for HP-UX. F*or 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

Agilent VISA is an I/O library that can be used to develop I/O applications and instrument drivers that comply with the VXI*plug&play* standards. Applications and instrument drivers developed with VISA can execute on VXI*plug&play* system frameworks that have the VISA I/O layer. Therefore, software from different vendors can be used together on the same system.

VISA Support on Windows

This 32-bit version of VISA is supported on Windows 95, Windows 98, Windows Me, Windows NT, and Windows 2000. (Support for the 16-bit version of VISA was removed in version H.01.00 of the Agilent IO Libraries.) 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), and LAN 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 Support on HP-UX

VISA is supported on the GPIB, VXI, GPIB-VXI, and LAN interfaces on HP-UX version 10.20. 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 95, Windows 98, Windows Me, Windows 2000, Windows NT, or HP-UX environment. To perform VISA installation and configuration on Windows NT or HP-UX, you must have system administration privileges on the Windows NT system or super-user (`root`) privileges on the HP-UX system.

VISA Documentation

This table shows associated documentation you can use when programming with Agilent VISA in the Windows or HP-UX environment.

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.
Agilent IO Libraries Installation and Configuration Guide for HP-UX	Shows how to install, configure, and maintain the Agilent IO Libraries on HP-UX.
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.
VXI*plug&play* System Alliance *VISA Library Specification 4.3*	Specifications for VISA.
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

Building a VISA Application in Windows

This chapter gives guidelines to build 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 gives guidelines to build 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 95, Windows 98, 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 NT or Windows 2000:

  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 file and library file search paths. They are set by:

- 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 file and library file 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 95, Windows 98, and Windows Me, the **idn** example files are in \Program Files\VISA\WIN95\AGVISA\SAMPLES.
- For VISA on Windows NT or Windows 2000, 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. If the program does not run, see the **Event Viewer** for a list of run-time errors.

```
          HEWLETT-PACKARD,54601A,0,1.7
```

```
/*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 example C/C++ program follows. For a more detailed explanation of VISA functionality, see *Chapter 4 - Programming With VISA.* See *Chapter 7 - VISA Language Reference* for more detailed information on these VISA function calls.

Function(s)	Description
visa.h	This file is included at the beginning of the file 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.
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 gives guidelines to build 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/NT, C:\Program Files\VISA\winnt\include**.** For Windows 95/98/Me, C:\Program Files\VISA\winnt\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 applicable directory
4. Click the **Open** button

VISA Limitations in Visual Basic

VISA functions return a status code which indicates success or failure of the function. The only indication of an error is the value of 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 = viVPrintf(vi, "%lf" + Chr$(10), dbl_value)
status = viVPrintf(vi, "%d" + Chr$(10), int_value)
```

Numeric Arrays

When reading to or writing from 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 95, Windows 98, or Windows Me, the **idn** example files are in C:\Program Files\VISA\WIN95\AGVISA\SAMPLES\ vb\idn.
For VISA on Windows NT or Windows 2000, the **idn** example files are in C:\Program Files\VISA\WINNT\AGVISA\SAMPLES\vb\idn.

Steps to Run the Program

The steps to build and run 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 and skip to Step 9.

3 Start a new Visual Basic Standard EXE project. VB 6.0 will open a new Project1 project 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 located in one of these directories:

\winnt\include\visa32.bas (Windows NT/2000)
\win95\include\visa32.bas (Windows 95/98/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 the 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 example Visual Basic program follows. For a more detailed explanation of VISA functionality, see *Chapter 4 - Programming with VISA*. See *Chapter 7 - VISA Language Reference* for more detailed information on these VISA function calls.

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 95/98/Me), the **Event Viewer** utility (for Windows 2000/NT), or the **Debug Window** (for Windows 95/98/2000/Me/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 or Windows 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 to use the **Event Viewer** is:

- Enable VISA logging from the Agilent IO Libraries Control, click **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 95, Windows 98, 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 to use the **Message Viewer** is:

- Enable VISA logging from the Agilent IO Libraries Control, click **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 to use the **Debug Window** is:
- Enable VISA logging from the Agilent IO Libraries Control. Click **VISA Logging | Debug Window**.
- Run your VISA program from Microsoft Visual Studio (or equivalent application).
- View error messages in the Visual Studio (or equivalent) output window.

3

Building a VISA Application in HP-UX

Building a VISA Application in HP-UX

This chapter gives guidelines to build a VISA application on HP-UX version 10.20 or later. The chapter contains the following sections:

- Building a VISA Program in HP-UX
- Using Online Help

Building a VISA Program in HP-UX

This section lists and example program called `idn` that queries a GPIB instrument for its identification string. The **`idn`** example program is located in the following subdirectory:

`opt/vxipnp/hpux/hpvisa/share/examples`

Example 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.

```
/*idn.c
   This program queries a GPIB device for an ID 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::24::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 sessions */
   viClose(vi);
   viClose(defaultRM);
}
```

Example Program Contents

A summary of the VISA function calls used in the example program follows. For a more detailed explanation of VISA functionality, see *Chapter 4 - Programming with VISA.* See *Chapter 7 - VISA Language Reference* for more detailed information on these VISA calls.

visa.h. This file is included at the beginning of the file 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.

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.

Running the Example Program

To run the **`idn`** example program, type the program name at the command prompt. For example:

```
idn
```

If the program run correctly, the following is an example of the output if connected to a 54601A oscilloscope:

```
Hewlett-Packard,54601A,0,1.7
```

If you have problems running the `idn` example program, first check to make sure the device address specified in your program is correct. If the program still does not run, check the I/O configuration. See the *Agilent I/O Libraries Installation and Configuration Guide for HP-UX* for information on I/O configuration.

Compiling and Linking a VISA Program

You can create your VISA applications in ANSI C or C++. When compiling and linking a C program that uses VISA, use the **`-lvisa`** command line option to link in the VISA library routines. The following example creates the **`idn`** executable file:

```
cc -Aa -o idn idn.c -lvisa
```

- The **`-Aa`** option indicates ANSI C
- The **`-o`** option creates an executable file called **`idn`**
- The **`-l`** option links in the VISA library

Logging Error Messages

To view any VISA internal errors that may occur on HP-UX, edit the **`/etc/opt/vxipnp/hpux/hpvisa.ini`** file. Change the `ErrorLog=` line in this file to the following:

```
ErrorLog=true
```

The error messages, if any, will be then be printed to **`stderr`**.

Using Online Help

Online help for VISA on HP-UX is provided with Bristol Technology's HyperHelp Viewer, or in the form of HP-UX manual pages (**man** pages), as explained in the following subsections.

Using the HyperHelp Viewer

The Bristol Technology HyperHelp Viewer allows you to view the VISA functions online. To start the HyperHelp Viewer with the VISA help file, type:

```
hyperhelp/opt/hyperhelp/visahelp.hlp
```

When you start the Viewer, you can also specify any of the following options

-k *keyword*	Opens the Viewer and searches for the specified *keyword*.
-p *partial_keyword*	Opens the Viewer and searches for a specific *partial keyword*.
-s *viewmode*	Opens the Viewer in the specified *viewmode*. If **1** is specified as the *viewmode*, the Viewer is shared by all applications. If **0** is specified, a separate Viewer is opened for each application (default).
-display *display*	Opens the Viewer on the specified *display.*

Using HP-UX Manual Pages

To use manual pages, type the HP-UX **man** command followed by the VISA function name:

```
man function
```

The following are examples of selecting online help on VISA functions:

```
man viPrintf
man viScanf
man viPeek
```

4

Programming with VISA

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 or in the examples subdirectory on HP-UX.

See *Appendix A - VISA Library Information* for the specific location of the example programs on your operating system. For specific details on VISA functions, see *Chapter 7 - VISA Language Reference.* 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 *Appendix B - VISA Resource Classes* 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 *Appendix B - VISA Resource Classes* for a description of the SERVANT resource.

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

Resource Class	Interface Types	Resource Class Description
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 *Appendix A - VISA System Information*.

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, you open a session with a specific resource with 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 by creating multiple session identifiers by calling the **viOpen** function multiple times. The following table summarizes the parameters in the previous 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).

Parameter	Description
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. (**M**ust 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 "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** is used for GPIB-VXI controllers.
- **TCPIP** establishes communication with LAN instruments.
- **VXI** is used for VXI instruments.

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**
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.

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.
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()*.
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.
VXI logical address	This is the logical address of the VXI instrument.

Some examples of valid symbolic names follow.

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.
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*.

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 "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);
```

Where the parameters are defined as follows.

Parameter	Description
sesn	The resource manager session.
expr	The expression that identifies what to search (see table that follows).
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.

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

Example: Searching 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 gives guidelines to send 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. See *Chapter 7 - VISA Language Reference* 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 *Chapter 7 - VISA Language Reference* 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 **[]**.

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.

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.

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).)

Argument Length Modifier	Format Codes	Description
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. Some example expressions follow the table.

Special Characters and Operators	Description
`?`	Matches any one character.
`\`	Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (e.g.,' '\?'), it matches the '?' character instead of any one character.
`[`*`list`*`]`	Matches any one character from the enclosed *list*. A hyphen can be used to match a range of characters.
`[^`*`list`*`]`	Matches any character not in the enclosed *list*. A hyphen can be used to match a range of characters.
`*`	Matches 0 or more occurrences of the preceding character or expression.
`+`	Matches 1 or more occurrences of the preceding character or expression.
***`exp`*`|`*`exp`**	Matches either the preceding or following expression. The or operator **`|`** matches the entire expression that precedes or follows it and not just the character that precedes or follows it. For example, **`VXI|GPIB`** means **`(VXI) | (GPIB)`**, not **`VXI(I|G)PIB`**.

Special Characters and Operators	Description
(*exp*)	Grouping characters or expressions.
" "	Sends a blank space.
\n	Sends the ASCII line feed character. The END identifier will also be sent.
\r	Sends an ASCII carriage return character.
\t	Sends an ASCII TAB character.
\###	Sends ASCII character specified by octal value.
\"	Sends the ASCII double quote character.
****	Sends a backslash character.

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**

Example Expression	Sample Matches
`(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.

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.

Format Codes	Description
[	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 *Chapter 7 - VISA Language Reference* 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.

This example program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See *Appendix A - VISA Library Information* 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 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 *Chapter 7 - VISA Language Reference.*

- **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.

```
/*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 gives guidelines to use 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..

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 Attributes	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 Attributes	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 Attributes	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_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attributes	Description	AP	Data Type	Range
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

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

.

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
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 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 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

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
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.

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

TCPIP Socket (SOCKET) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attributes	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

TCPIP Socket (SOCKET) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attributes	Description	AP	Data Type	Range
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

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. Then, when the enabled event occurs, the installed event handler is called.

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

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 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*`);`

Where the parameters are defined as follows:

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 *Chapter 7 - VISA Language Reference* 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.

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

This example program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See *Appendix A - VISA Library Information* for locations of example programs on your operating system.

```
/* 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.

This program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See *Appendix A - VISA Library Information* for locations of example programs on your operating system.

```
/* 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);
```

where the parameters are defined as follows:

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.

This example program is installed on your system in the SAMPLES subdirectory on Windows environments or in the examples subdirectory on HP-UX. See *Appendix A - VISA Library Information* for locations of example programs on your operating system.

```
/* 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 gives guidelines to trap 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: Check 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 applicationfrom 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.

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 *Chapter 7 - VISA Language Reference* 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 also 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 *Chapter 7 - VISA Language Reference* 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);
}
```

Notes:

5

Programming via GPIB and VXI

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 4 - Programming with VISA* for general information on VISA programming for the GPIB, VXI, and GPIB-VXI interfaces. See *Chapter 7 - VISA Language Reference* 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
- GPIB-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 IO Interface?

An **IO 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 an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **`viOpen`** function call in a VISA program.

IO Config assigns an Interface Name and Logical Unit Number 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).

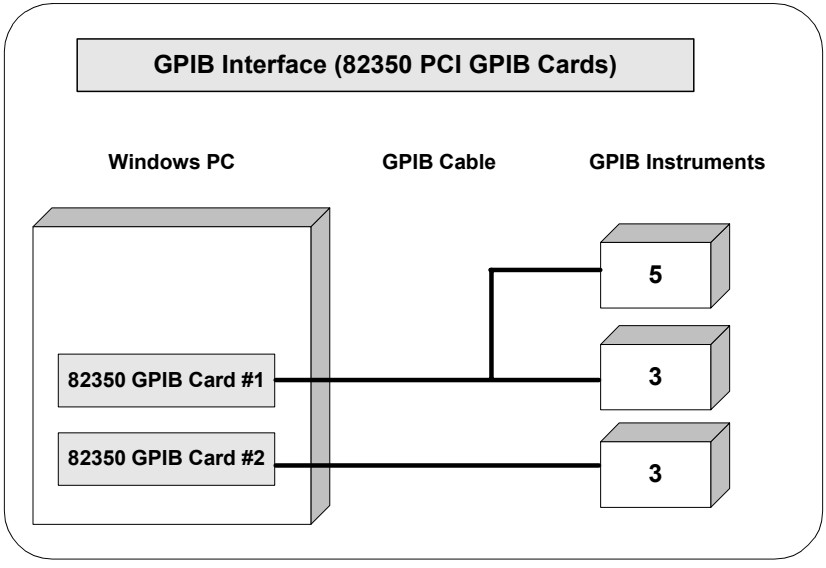
A **message-based VXI device** has its own processor that allows it to interpret high-level commands such as Standard Commands for Programmable Instruments (SCPI). When using VISA, you can place the SCPI command within your VISA output function call. Then, the message-based device interprets the SCPI 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.

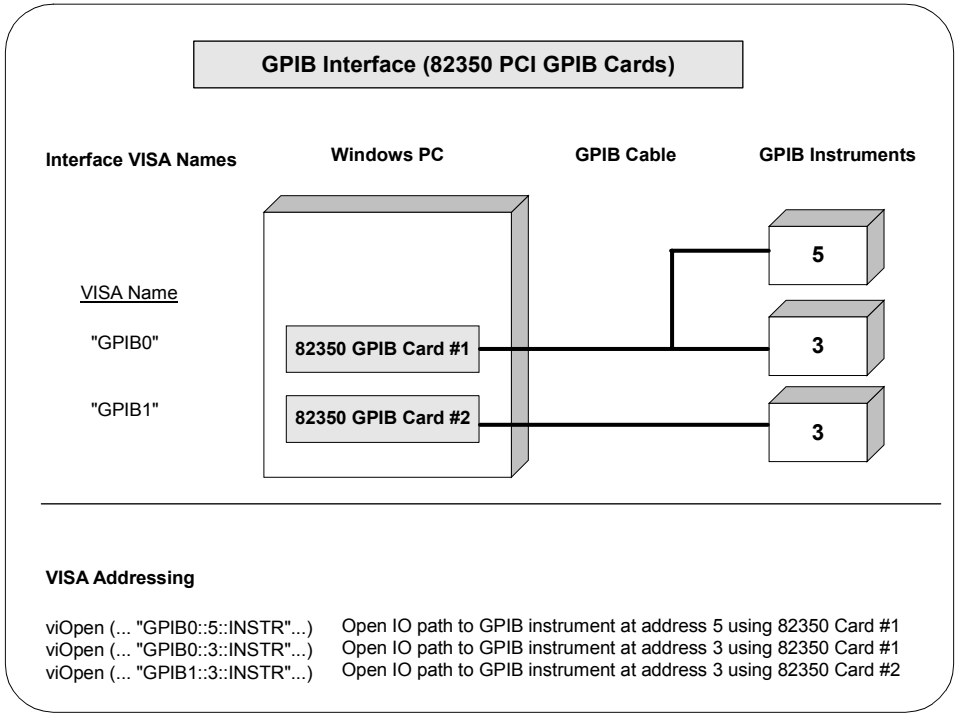
GPIB Interfaces Overview

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. This figure shows GPIB instruments at addresses 3 and 5.



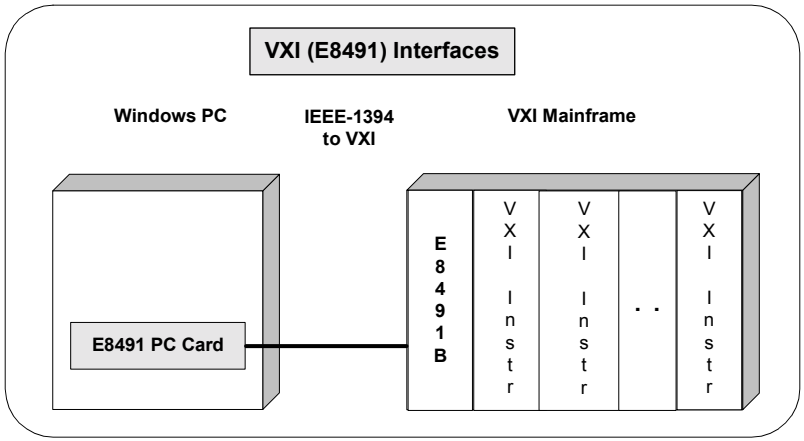
Example: GPIB (82350) Interface

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.



VXI Interfaces Overview

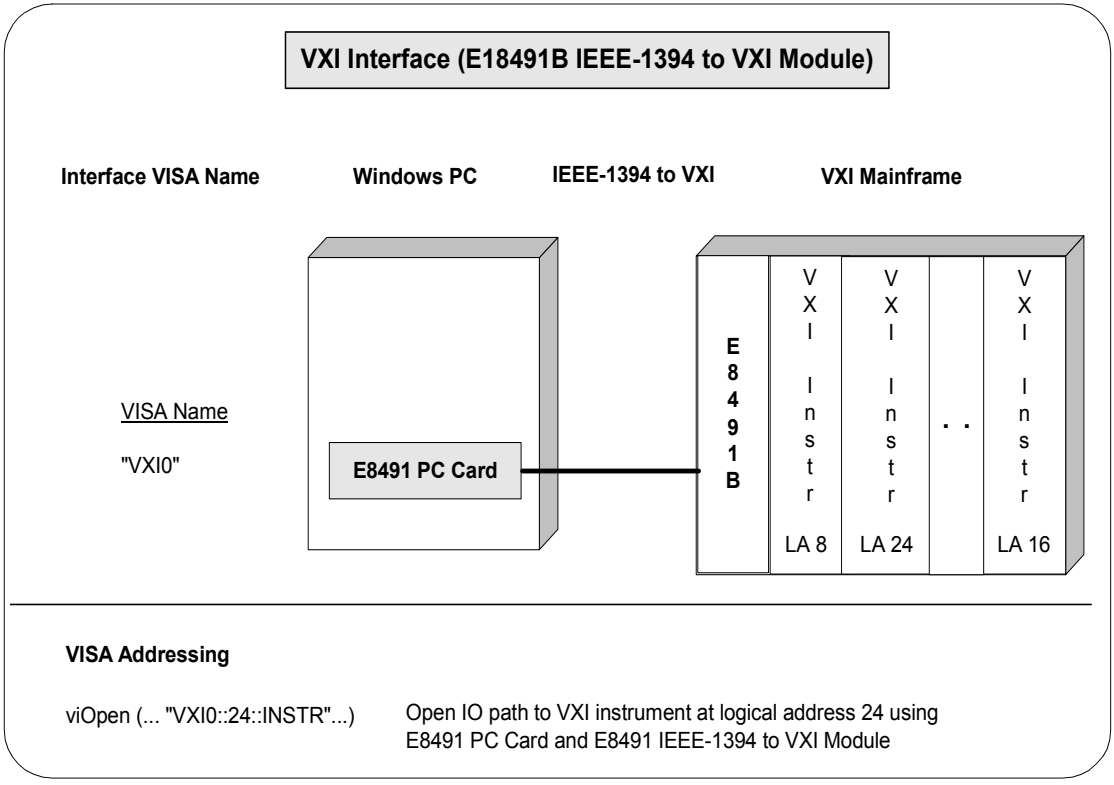
As shown in the following figure, a typical VXI (E8491) interface consists of an E8491 PC Card in a Windows PC that is connected to an E8491B IEEE-1394 Module in a VXI mainframe via an IEEE-1394 to VXI cable. The VXI mainframe also includes one or more VXI instruments.



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.*

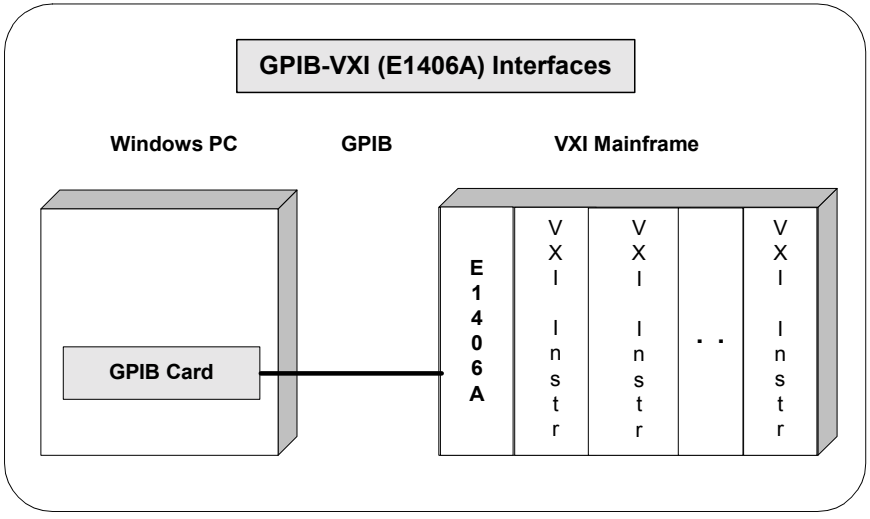

VXI Interface (E18491B IEEE-1394 to VXI Module)
Interface VISA Name
Windows PC
IEEE-1394 to VXI
VXI Mainframe
VISA Name
"VXI0"
E8491 PC Card
E8491B
VXI Instr
LA 8
VXI Instr
LA 24
. .
VXI Instr
LA 16
VISA Addressing
viOpen (... "VXI0::24::INSTR"...)
Open IO path to VXI instrument at logical address 24 using
E8491 PC Card and E8491 IEEE-1394 to VXI Module

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.



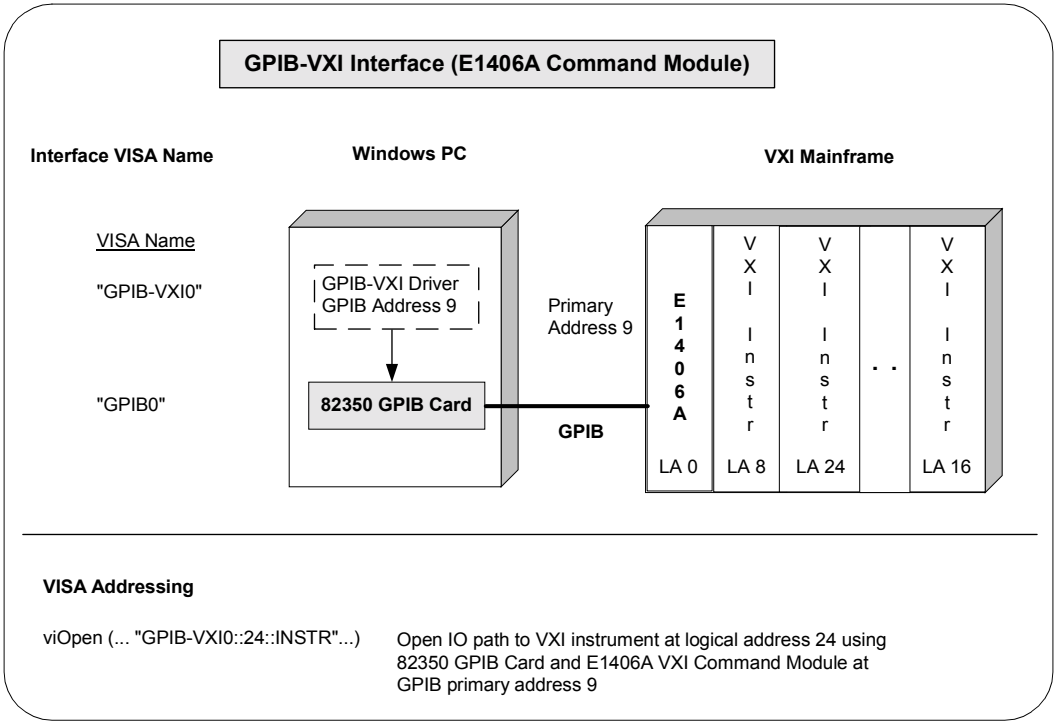
Example: GPIB-VXI (E1406A) Interface

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”.

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.

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.
`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.

Function	Description
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 the 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 the 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.

Function	Description
viMapAddress(*vi, mapSpace, mapBase, mapSize, access, suggested, address***);**	Maps the specified memory space.
viPeek8(*vi, addr, val8***);**	Reads 8 bits of data from address specified.
viPeek16(*vi, addr, val16***);**	Reads 16 bits of data from address specified.

Function	Description
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 are:

```
/* 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 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 dereferenced 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 access 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 *Appendix B - VISA Resource Classes* 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 EXIT   1
#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;
   ViUInt16  inData16 = 0;
   ViUInt16  peekData16 = 0;
   ViUInt8   *addr;
   ViUInt16  *addr16;
   ViStatus  status;
   ViUInt16  offset;

   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 retruned 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.

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.

Attribute	Description
VI_ATTR_VXI_LA	Logical address of the local controller.
VI_ATTR_SRC_INCREMENT	Used in **viMoveIn*xx*** operation to specify how much the source offset is to be incremented after every transfer. The default value is 1 and the **viMoveIn*xx*** operation moves from consecutive elements. If this attribute is set to 0, the **viMoveIn*xx*** operation will always read from the same element, essentially treating the source as a FIFO register.

Attribute	Description
`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.
`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 address modifier used in high-level access operations such as **`viOutxx`** and **`viMoveOutxx`**, when writing to destination.
`VI_ATTR_WIN_ACCESS_PRIV`	Specifies 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.

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.
`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.

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 *Appendix B - VISA Resource Classes* 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:

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

This example shows how you can read the **VI_ATTR_WIN_ACCESS** attribute and use the result to determine how to access memory.

```
ViAddr address;
ViUInt16 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.*)

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 4 - 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.

Notes:

6

Programming via LAN

Programming via LAN

This chapter gives 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 GPIB Devices via LAN

NOTE

This chapter does not describe programming using the VISA TCPIP Interface Type. To use GPIB over the LAN, you must first configure the TCPIP:LAN Client interface and then the VISA LAN Client during Agilent IO Libraries configuration.

The TCPIP VISA interface type can be used directly to communicate with GPIB devices over LAN, without having to configure a VISA LAN Client.

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.

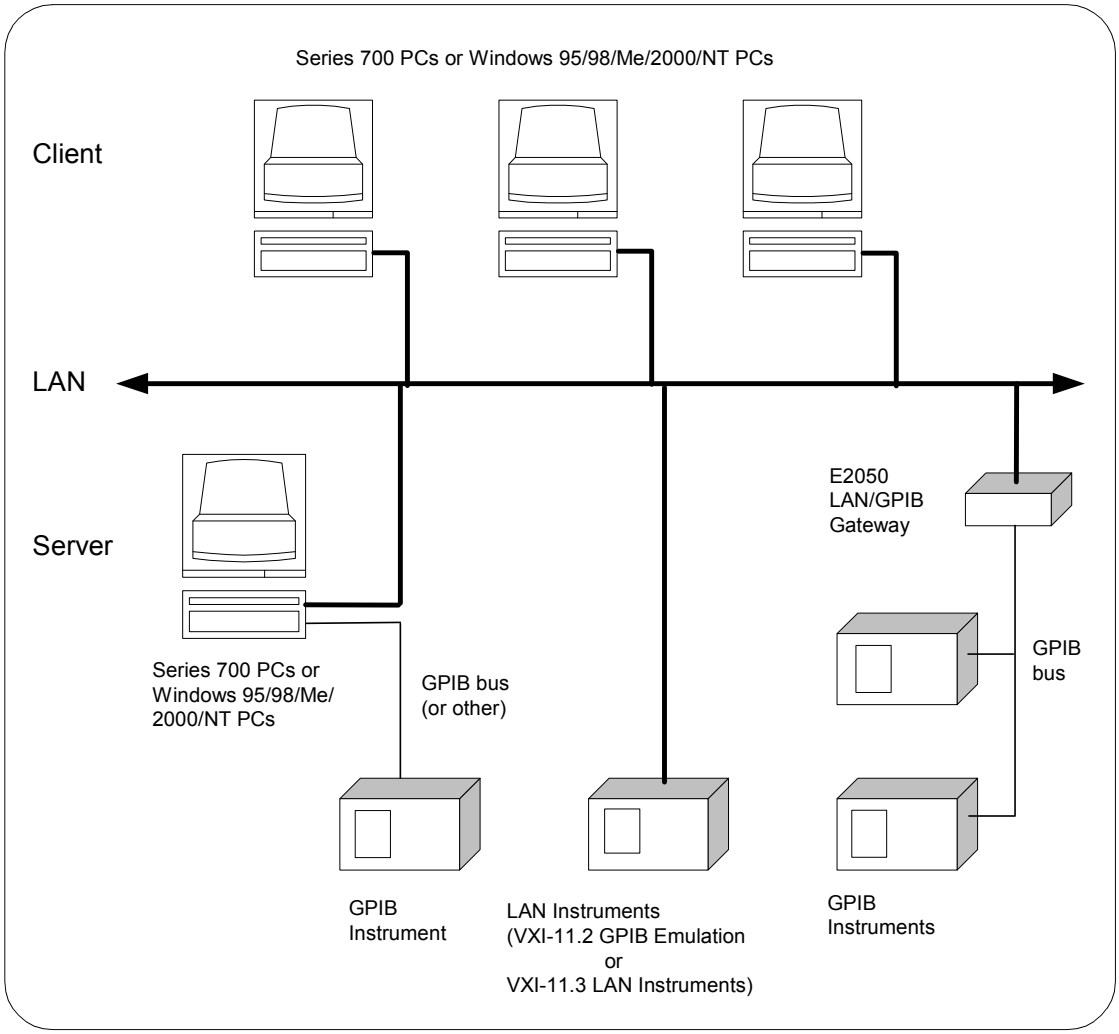
Client/Server Model

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 95/98/Me/NT/2000 PC) makes VISA requests over the network to a LAN server (such as a Series 700 HP-UX workstation, a Windows 95/98/Me/NT/2000 PC, or an E2050 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.

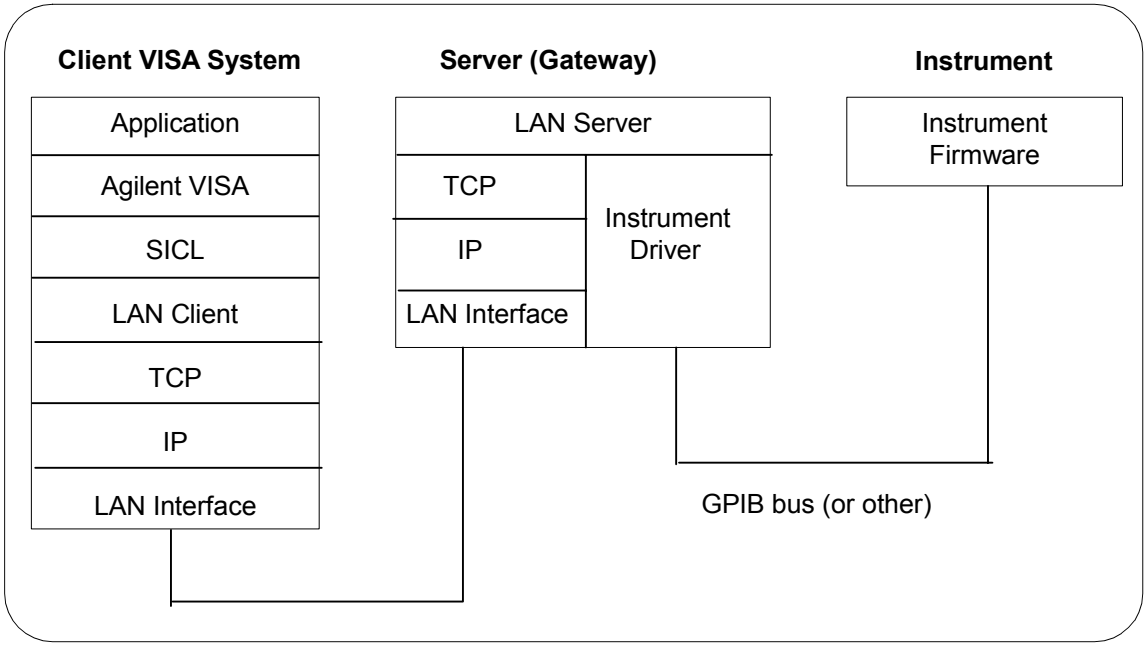

Series 700 PCs or Windows 95/98/Me/2000/NT PCs
Client
LAN
Server
E2050
LAN/GPIB
Gateway
GPIB
bus
Series 700 PCs or
Windows 95/98/Me/
2000/NT PCs
GPIB bus
(or other)
GPIB
Instrument
LAN Instruments
(VXI-11.2 GPIB Emulation
or
VXI-11.3 LAN Instruments)
GPIB
Instruments

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 an **Interface Name** or **Logical Unit Number** to identify an interface. This information is passed in the parameter string of the **`viOpen`** function call in a VISA program.

IO Config assigns an Interface Name and Logical Unit Number 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 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 HP-UX 10.20, Windows 95/98/Me/2000/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 protocols 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, an HP Series 700 computer running HP-UX, or a PC running Windows 95/98/Me/2000/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:

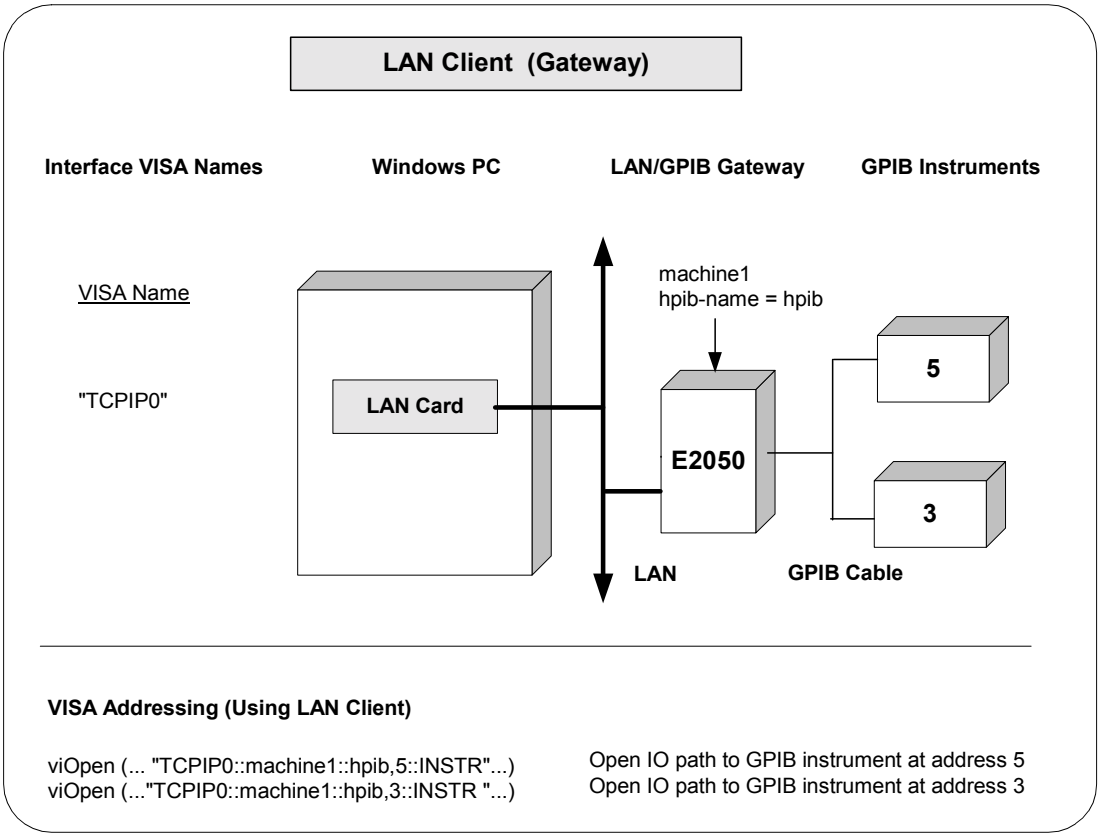
- LAN Client (Gateway)
- LAN Client (LAN)

This section provides an example of each configuration and shows applicable VISA **`viOpen`** commands. See *Chapter 7 - VISA Language Reference* 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 E2050 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 in the figure.

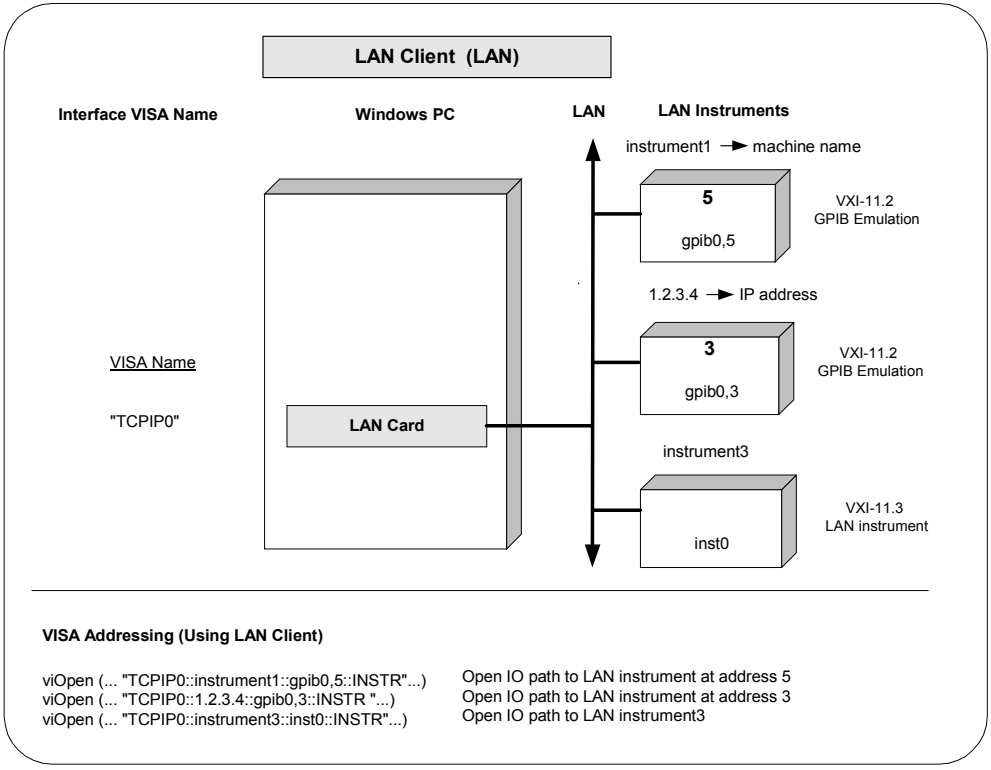


Example: LAN Client (LAN) Interface

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 *Chapter 7 - VISA Language Reference* for details on the VISA commands.

NOTE

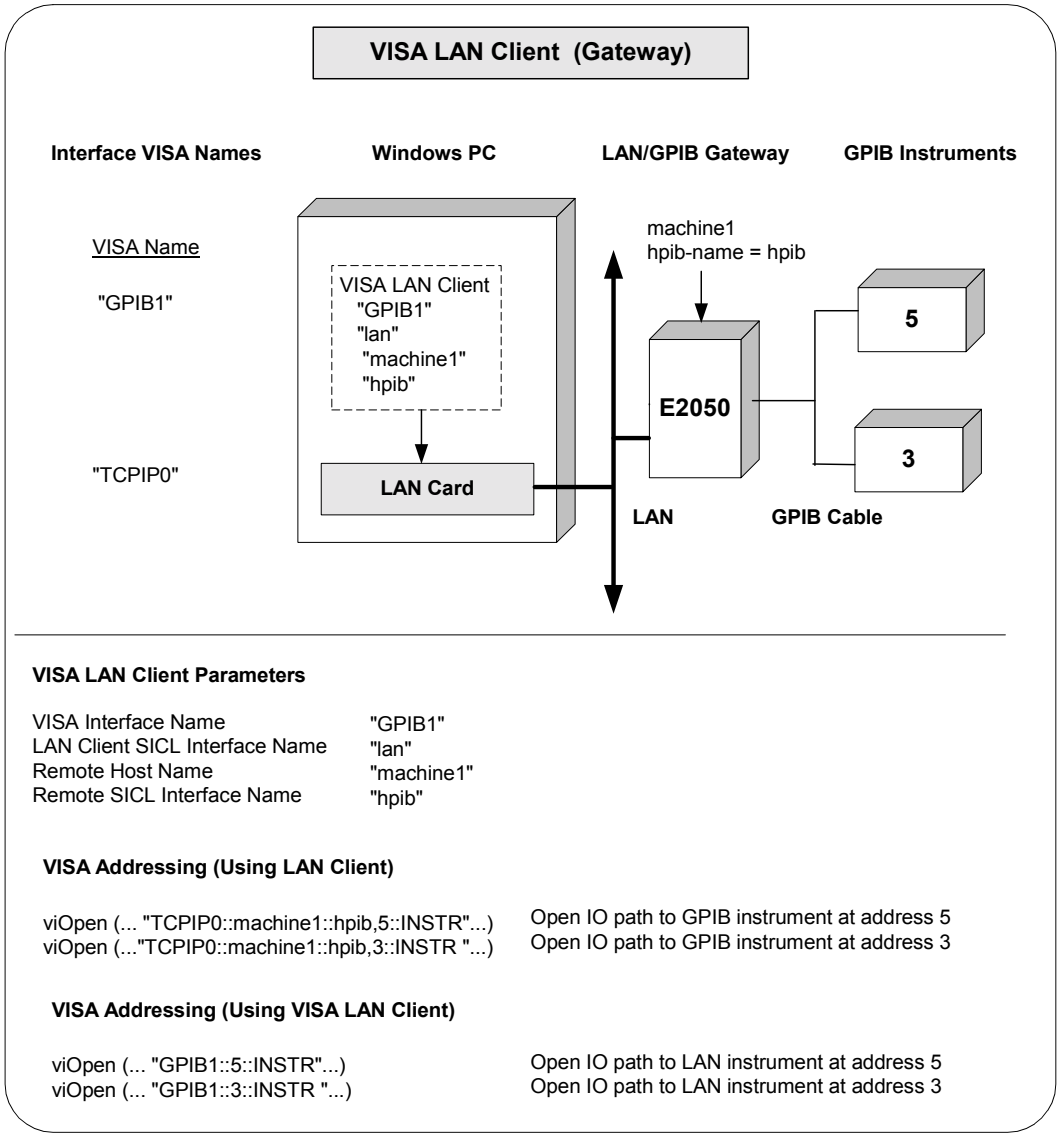
You must install a LAN Client interface BEFORE you can use a VISA LAN Client interface. See “Configuring LAN Client Interfaces” for details on configuring LAN Client interfaces.

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 E2050 LAN/GPIB gateway, and two GPIB instruments. The IO Config utility has been used to assign the LAN card a VISA name of “TCPIP0”.

In addition, a VISA LAN Client has been configured with the interface names and host names shown in the figure. Also, the E2050 LAN/GPIB Gateway has been assigned a name of machine1 and an hpib-name = hpib.

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 Parameters

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

VISA Addressing (Using LAN Client)

viOpen (... "TCPIP0::machine1::hpib,5::INSTR"...)	Open IO path to GPIB instrument at address 5
viOpen (..."TCPIP0::machine1::hpib,3::INSTR "...)	Open IO path to GPIB instrument at address 3

VISA Addressing (Using VISA LAN Client)

viOpen (... "GPIB1::5::INSTR"...)	Open IO path to LAN instrument at address 5
viOpen (... "GPIB1::3::INSTR "...)	Open IO path to LAN instrument at address 3

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.

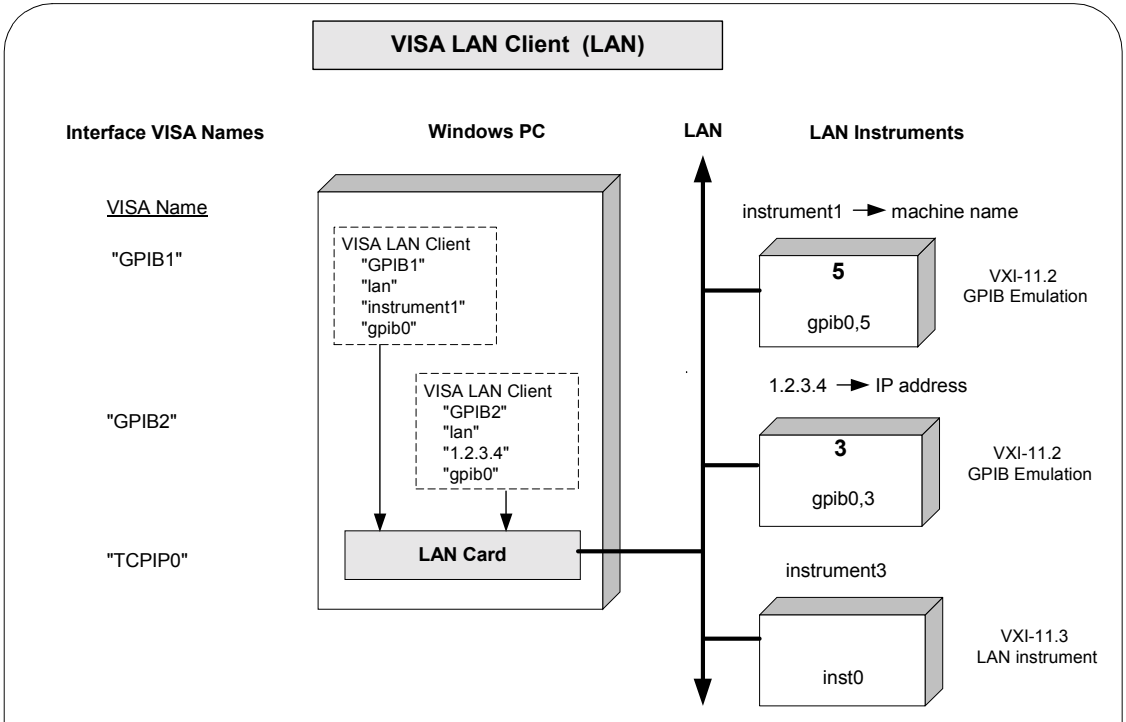
For this system, the IO Config utility has been used to assign the LAN card a VISA name of “TCPIP0”. In addition, two VISA LAN Clients have been configured with the interface names and host names shown in the figure.

For the addressing examples, instrument1 has been addressed by its 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. Note, however, that you cannot talk to instrument3 with 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 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	

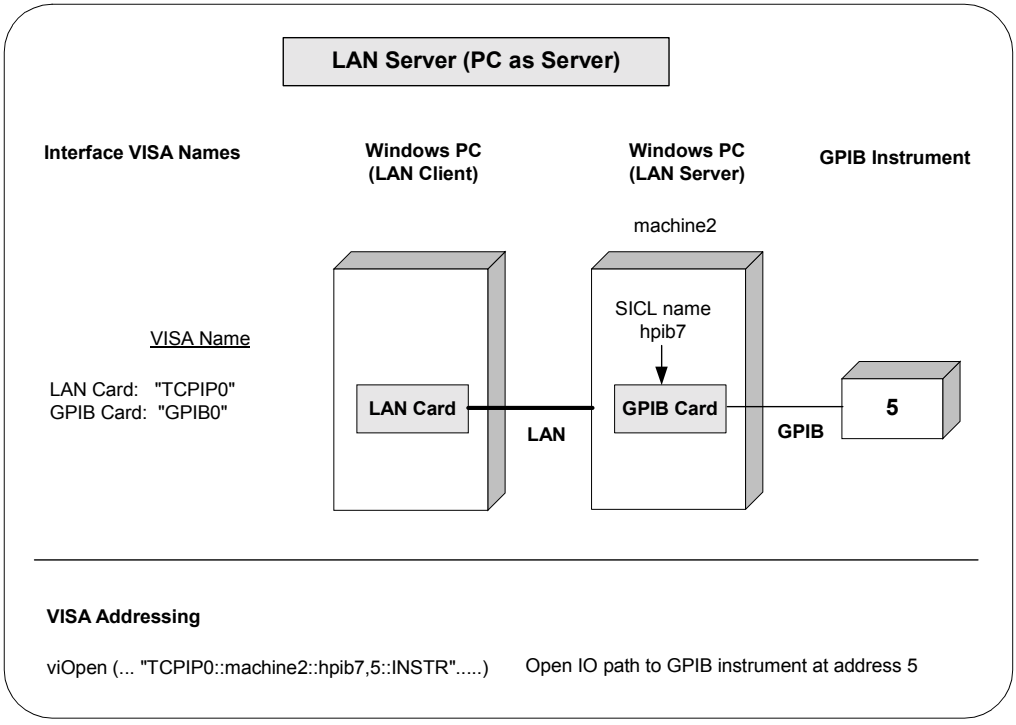
LAN Server Interface Overview

This section provides an example of the LAN Server interface configuration and shows applicable VISA **`viOpen`** commands. See *Chapter 7 - VISA Language Reference* 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, and a GPIB instrument. The IO Config utility has been used to assign the LAN card a VISA name of “TCPIP0”. Also, the GPIB card in the LAN server PC has been assigned VISA name of “GPIB0”. The LAN server PC has been assigned a name of machine2.

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.



Communicating with GPIB Devices via LAN

VISA supports LAN-gatewayed sessions to communicate with configured LAN servers. Since the LAN server configuration is determined by the type of server present, the only action required by the user is to configure VISA for a VISA LAN Client during Agilent IO Libraries configuration. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on configuring a VISA LAN Client.

NOTE

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

Addressing a Session

In general, the rules to address a LAN session are the same as to address a GPIB 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. This example illustrates addressing a GPIB device configured over the LAN.

GPIB0::7::0	A GPIB device at primary address 7 and secondary address 0 on the GPIB interface. This GPIB interface (GPIB0) is configured as a VISA LAN Client in the Agilent IO Libraries configuration.

Example: Opening a Device Session

This example shows one way to open a device session with a GPIB device at primary address 23. See *Chapter 4 - Programming with VISA* for more information on addressing device sessions.

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

Example: LAN Session

This program opens a session with a GPIB device and sends a comma operator to send a comma-separated list. The 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 *Chapter 4 - Programming with VISA* for information on error trapping.

```
/*formatio.c
   This example program makes a multimeter measurement
   with a comma-separated list passed with formatted
   I/O and prints the results. Note that you must 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(&defaultRM);
   viOpen(defaultRM, "GPIB0::22::INSTR", VI_NULL,
          VI_NULL, &vi);

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

   /* Set up device and send 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 Timeouts over LAN

The client/server architecture of the LAN software requires the use of two timeout values: one for the client and one for the server.

Client/Server Operation

The server's timeout value is specified by setting a VISA timeout via the **`VI_ATTR_TMO_VALUE`** attribute. The server will also adjust the requested value if infinity is requested. The client's timeout value is determined by the values set when you configure the **`LAN Client`** during the Agilent IO Libraries configuration. See the *Agilent IO Libraries Installation and Configuration Guide* for configuration information.

When the client sends an I/O request to the server, the timeout value determined by the values set with the **`VI_ATTR_TMO_VALUE`** attribute is passed with the request. The client may also adjust the value sent to the server if **`VI_TMO_INFINITE`** was specified. The server will use that timeout in performing the I/O operation, just as if that timeout value had been used on a local I/O operation.

If the server's operation is not complete in the specified time, the server will send a reply to the client which indicates that a timeout occurred, and the VISA call made by the application will return an error.

When the client sends an I/O request to the server, it starts a timer and waits for the reply from the server. If the server does not reply in the time specified, the client stops waiting for the reply from the server and returns an error.

LAN Timeout Values

The LAN Client configuration specifies two timeout-related configuration values for the LAN software. These values are used by the software to calculate timeout values. See the *Agilent IO Libraries Installation and Configuration Guide for Windows* for information on setting these values.

- **Server Timeout**. Timeout value passed to the server when an application sets the VISA timeout to infinity (**`VI_TMO_INFINITE`**). Value specifies the number of seconds the server will wait for the operation to complete before returning an error. If this value is zero (0), the server will wait forever.
- **Client Timeout Delta.** Value added to the VISA timeout value (server's timeout value) to determine the LAN timeout value (client's timeout value). Value specifies the number of seconds.

The timeouts are adjusted using the following algorithm:

- The VISA Timeout, which is sent to the server for the current call, is adjusted if it is currently infinity (**`VI_TMO_INFINITE`**). In that case, it will be set to the Server Timeout value.
- The LAN Timeout is adjusted if the VISA Timeout plus the Client Timeout Delta is greater than the current LAN Timeout. In this case, the LAN Timeout is set to the VISA Timeout plus the Client Timeout Delta.
- The calculated LAN Timeout increases as necessary to meet the needs of the application, but never decreases. This avoids the overhead of readjusting the LAN Timeout every time the application changes the VISA Timeout.

To change the defaults:

1. Run the IO Config utility (Windows) or the visacfg utility (HP-UX).
2. Edit the LAN Client interface.
3. Change the Server Timeout or Client Timeout Delta parameter. (See online help for information on changing these values.)
4. Restart the VISA LAN applications.

Application Terminations and Timeouts

If an application is killed either via **Ctrl**+**C** or the HP-UX **`kill`** command during a VISA operation performed at the LAN server, the server will continue to try the operation until the server's timeout is reached.

By default, the LAN server associated with an application using a timeout of infinity that is killed may not discover that the client is no longer running for up to two minutes. (If you are using a server other than the LAN server supported with the product, check that server's documentation for its default behavior.)

If both the LAN client and LAN server are configured to use a long timeout value, the server may appear "hung." If this situation is encountered, the LAN client (via the Server Timeout value) or the LAN server may be configured to use a shorter timeout value.

If long timeouts must be used, the server may be reset. An HP-UX server may be reset by logging into the server host and killing the running **`sicllan`**d daemon(s). However, this procedure will affect all clients connected to the server.

A Windows 95, Windows 98, Windows Me, Windows 2000, or Windows NT server may be reset by typing **Ctrl**+**C** in the LAN Server window and then restarting the server from the Agilent IO Libraries program group. This procedure will also affect all clients connected to the server.

LAN Signal Handling on HP-UX

This section describes how to use signal handling and service requests over LAN for HP-UX.

Using Signal Handling over LAN (HP-UX Only)

VISA uses SIGIO signals for SRQs on LAN interfaces on HP-UX. The VISA LAN Client installs a signal handler to catch SIGIO signals. To enable sharing of SIGIO signals with other portions of an application, the VISA LAN SIGIO signal handler remembers the address of any previously installed SIGIO handler and calls this handler after processing a SIGIO signal itself.

If your application installs a SIGIO handler, it should also remember the address of a previously installed handler and call it before completing. The signal number used with LAN (SIGIO) *cannot* be changed.

Notes:

7

VISA Language Reference

VISA Language Reference

This chapter describes each function in the VISA library for the Windows and HP-UX programming environments and provides an alphabetical list of interfaces and Resource Classes associated with each functions. VISA functions are listed in alphabetical order.

VISA Functions Overview

This section lists VISA functions by applicable interfaces and resource classes, and lists VISA functions by type of operations performed.

VISA Functions by Interface/Resource

This table lists VISA functions, supported interfaces (GPIB, VXI, etc.) and associated resource classes (INSTR, INTFC, etc.) that are implemented in Agilent VISA.

Interface	---	---	GPIB		GPIB-VXI	VXI			TCPIP		ASRL
Resource Class	Find List	Rsrc-Mgr	INSTR	INTFC	INSTR	INSTR	MEM-ACC	BACK-PLANE	INSTR	SOC-KET	INSTR
`viAssertIntrSignal`											
`viAssertTrigger`			•	•	•	•		•	•	•	•
`viAssertUtilSignal`											
`viBufRead`			•	•	•	•		•	•	•	•
`viBufWrite`			•	•	•	•		•	•	•	•
`viClear`			•		•	•		•	•	•	•
`viClose`	•	•	•	•	•	•	•	•	•	•	•
`viDisableEvent`		•	•	•	•	•	•	•	•	•	•
`viDiscardEvents`		•	•	•	•	•	•	•	•	•	•
`viEnableEvent`		•	•	•	•	•	•	•	•	•	•
`viEventHandler`			•		•	•		•	•	•	•
`viFindNext`	•		•		•	•		•	•	•	•
`viFindRsrc`		•	•		•	•		•	•	•	•
`viFlush`			•		•	•		•	•	•	•
`viGetAttribute`	•	•	•		•	•		•	•	•	•

Interface	---	---	GPIB		GPIB-VXI	VXI			TCPIP		ASRL
Resource Class	Find List	Rsrc-Mgr	INSTR	INTFC	INSTR	INSTR	MEM-ACC	BACK-PLANE	INSTR	SOC-KET	INSTR
`viGpibCommand`				•							
`viGpibControlATN`				•							
`viGpibControlREN`			•	•					•		
`viGpibPassControl`				•							
`viGpibSendIFC`				•							
`viIn8`					•	•	•				
`viIn16`					•	•	•				
`viIn32`					•	•	•				
`viInstallHandler`		•	•		•	•		•	•	•	•
`viLock`			•		•	•		•	•	•	•
`viMapAddress`					•	•	•				
`viMapTrigger`								•			
`viMemAlloc`											
`viMemFree`											
`viMove`					•	•	•				
`viMoveAsync`[a]					•	•	•				
`viMoveIn8`					•	•	•				
`viMoveIn16`					•	•	•				
`viMoveIn32`					•	•	•				
`viMoveOut8`					•	•	•				

a = implemented synchronously

Interface	---	---	GPIB		GPIB-VXI	VXI			TCPIP		ASRL
Resource Class	Find List	Rsrc-Mgr	INSTR	INTFC	INSTR	INSTR	MEM-ACC	BACK-PLANE	INSTR	SOC-KET	INSTR
viMoveOut16					•	•	•				
viMoveOut32					•	•	•				
viOpen		•	•		•	•		•	•	•	•
viOpenDefaultRM		•	•		•	•		•	•	•	•
viOut8					•	•	•				
viOut16					•	•	•				
viOut32					•	•	•				
viParseRsrc		•	•		•	•		•	•	•	•
viPeek8					•	•	•				
viPeek16					•	•	•				
viPeek32					•	•	•				
viPoke8					•	•	•				
viPoke16					•	•	•				
viPoke32					•	•	•				
viPrintf			•	•	•				•	•	•
viQueryf			•		•	•			•		•
viRead			•	•	•	•			•	•	•
viReadAsync[a]			•	•	•	•			•	•	•
viReadSTB			•		•	•			•	•	•
viReadToFile			•	•	•	•			•	•	•

a = implemented synchronously

Interface	---	---	GPIB		GPIB-VXI	VXI			TCPIP		ASRL
Resource Class	Find List	Rsrc-Mgr	INSTR	INTFC	INSTR	INSTR	MEM-ACC	BACK-PLANE	INSTR	SOC-KET	INSTR
viScanf			•	•	•	•			•	•	•
viSetAttribute	•	•	•		•	•		•	•	•	•
viSetBuf			•	•	•	•			•	•	•
viSPrintf			•	•	•	•			•	•	•
viSScanf			•	•	•	•			•	•	•
viStatusDesc	•	•	•		•	•		•	•	•	•
viTerminate											
viUninstallHandler		•	•		•	•		•	•	•	•
viUnlock			•		•	•		•	•	•	•
viUnmapAddress					•	•	•				
viUnmapTrigger					•	•	•				
viVPrintf			•	•	•	•			•	•	•
viVQueryf			•		•	•			•		•
viVScanf			•	•	•	•			•	•	•
viVSPrintf			•	•	•	•			•	•	•
viVSScanf			•	•	•	•			•	•	•
viVxiCommandQuery						•					
viWaitOnEvent		•	•		•	•		•	•	•	•
viWrite			•	•	•	•			•	•	•
viWriteAsync			•	•	•	•			•	•	•
viWriteFromFile			•	•	•	•			•	•	•

VISA Functions by Type

This table show VISA functions implemented by Agilent VISA grouped by type. The data types for the VISA function parameters (for example, **ViSession**, etc.) are defined in the VISA declarations file (see *Appendix A - VISA Library Information*).

Operation	Function (Type *Parameter1*, Type *Parameter2*, ...);
Opening/Closing Sessions	
Open Default RM Session	`viOpenDefaultRM(ViSession sesn);`
Open Session	`viOpen(ViSession sesn, ViRsrc rsrcName, ViAccessMode accessMode, ViUInt32 timeout, ViSession vi);`
Close Session	`viClose(ViSession/ViEvent/ViFindList vi);`
Control	
Get Attribute	`viGetAttribute(ViSession/ViEvent/ViFindList vi, ViAttr attribute, ViAttrState attrState);`
Set Attribute	`viSetAttribute(ViSession/ViEvent/ViFindList vi, ViAttr attribute, ViAttrState attrState);`
Get Status Code Description	`viStatusDesc(ViSession/ViEvent/ViFindList vi, ViStatus status, ViString desc);`
Terminate Asynchronous Operation	`viTerminate(ViSession vi, ViUInt16 degree, ViJobId jobId);`
Lock Resource	`viLock(ViSession vi, ViAccessMode lockType, ViUInt32 timeout, ViKeyId requestedKey, ViKeyId accessKey);`
Unlock Resource	`viUnlock(ViSession vi);`
Map Trigger Source Line to Destination Line	`viMapTrigger(ViSession vi, ViInt16 trigSrc, ViInt16 trigDest, ViUInt16 mode);`
Map Trigger Line to Another Trigger Line	`viUnmapTrigger(ViSession vi, ViInt16 trigSrc, ViInt16 trigDest);`

Operation	Function (Type *Parameter1*, Type *Parameter2*, ...);
Event Handling/Interrupts	
Enable Event	`viEnableEvent(ViSession vi, ViEventType eventType, ViUInt16 mechanism, ViEventFilter context);`
Disable Event	`viDisableEvent(ViSession vi, ViEventType eventType, ViUInt16 mechanism);`
Discard Events	`viDiscardEvents(ViSession vi, ViEventType eventType,ViUInt16 mechanism);`
Wait on Event	`viWaitOnEvent(ViSession vi, ViEventType inEventType,ViUInt32 timeout, ViEventType outEventType, ViEvent outContext);`
Install Handler	`viInstallHandler(ViSession vi, ViEventType eventType, ViHndlr handler, ViAddr userHandle);`
Uninstall Handler	`viUninstallHandler(ViSession vi, ViEventType eventType,ViHndlr handler, ViAddr userHandle);`
Event Handler Prototype	`viEventHandler(ViSession vi, ViEventType eventType, ViEvent context, ViAddr userHandle);`
VXI Specific Series	
Send Device a Command/Query and/or Retrieve a Response	`viVxiCommandQuery(ViSession vi, ViUInt16 mode, ViUInt32 cmd, ViUInt32 response);`
Searching	
Find Device	`viFindRsrc(ViSession sesn, ViString expr, ViFindList findList, ViUInt32 retcnt, ViRsrc instrDesc);`
Find Next Device	`viFindNext(ViFindList findList, ViRsrc instrDesc);`
Parse Resource String to Get Interface Information	`viParseRsrc(ViSession sesn, ViRsrc rsrcName, VIUInt16 intfType, VIUInt16 intfNum);`
Basic I/O	
Read Data from Device	`viRead(ViSession vi, ViBuf buf, ViUInt32 count, ViUInt32 retCount);`
Write Data to Device	`viWrite(ViSession vi, ViBuf buf, ViUInt32 count, ViUInt32 retCount);`
Read Data Asynchronously from Device	`viReadAsync(ViSession vi, ViBuf buf, ViUInt32 count, ViJobId jobId);`
Write Data Asynchronously to Device	`viWriteAsync(ViSession vi, ViBuf buf, ViUInt32 count, ViJobId jobId);`

Operation	Function (Type *Parameter1*, Type *Parameter2*, ...);
Basic I/O (continued)	
Clear a Device	`viClear(ViSession vi);`
Read Data Synchronously and Store Data in File	`viReadToFile (ViSession vi, ViConstString fileName, ViUInt32 count, ViUInt32 retCount);`
Write Data from File Synchronously	`viWriteFromFile (ViSession vi, ViConstString fileName, ViUInt32 count, ViUInt32 retCount);`
Assert Software/Hardware Trig	`viAssertTrigger(ViSession vi, ViUInt16 protocol);`
Read Status Byte	`viReadSTB(ViSession vi, ViUInt16 status);`
Formatted I/O	
Set Size of Buffer	`viSetBuf(ViSession vi, ViUInt16 mask, ViUInt32 size);`
Unformatted Read to Formatted I/O Buffers	`viBufRead(ViSession vi, ViBuf buf, ViUInt32 count, ViUInt32 retCount);`
Unformatted Write to Formatted I/O Buffers	`viBufWrite (ViSession vi, ViBuf buf, ViUInt32 count, ViUInt32 retCount);`
Flush Read and Write Buffers	`viFlush(ViSession vi, ViUInt16 mask);`
Convert, Format, and Send Parameters to a Device	`viPrintf(ViSession vi, ViString writeFmt, arg1,arg2, ...);`
Convert, Format, and Send Parameters to a Device	`viVPrintf(ViSession vi, ViString writeFmt, ViVAList params);`
Write Data to a Buffer	`viSPrintf(ViSession vi, ViBuf buf, ViString writeFmt, arg1, arg2, ...);`
Convert, Format, and Send Parameters to a Buffer	`viVSPrintf(ViSession vi, ViBuf buf, ViString writeFmt, ViVAList params);`
Receive Data from Device, Format and Store Data	`viScanf(ViSession vi, ViString readFmt, arg1, arg2, ...);`
Receive Data from Device, Format and Store Data	`viVScanf(ViSession vi, ViString readFmt, ViVAList params);`
Receive Data from Buffer, Format and Store Data	`viSScanf(ViSession vi, ViBuf buf, ViString readFmt, arg1, arg2, ...);`
Receive Data from Buffer, Format and Store Data	`viVSScanf(ViSession vi, ViBuf buf, ViString readFmt, ViVAList params);`

Operation	Function (Type *Parameter1*, Type *Parameter2*, ...);
Formatted I/O (continued)	
Formatted Write and Read Operation	`viQueryf(ViSession vi, ViString writeFmt, ViString readFmt, arg1, arg2, ...);`
Formatted Write and Read Operation	`viVQueryf(ViSession vi, ViString writeFmt, ViString readFmt, ViVAList params);`
Memory I/O	
Read 8-bit Value from Memory Space	`viIn8(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt8 val8);`
Read 16-bit Value from Memory Space	`viIn16(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt16 val16);`
Read 32-bit Value from Memory Space	`viIn32(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt32 val32);`
Write 8-bit Value to Memory Space	`viOut8(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt8 val8);`
Write 16-bit Value to Memory Space	`viOut16(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt16 val16);`
Write 32-bit Value to Memory Space	`viOut32(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt32 val32);`
Move data from source to destination	`viMove (ViSession vi, ViUInt16 srsSpace, ViBusAddress srcOffset, ViUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress destOffset, ViUInt16 destWidth, ViBusSize length)`
Move data from source to destination asynchronously	`viMoveAsync (ViSession vi, ViUInt16 srsSpace, ViBusAddress srcOffset, ViUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress destOffset, ViUInt16 destWidth, ViBusSize length, ViJobId jobId)`
Move 8-bit Value from Device Memory to Local Memory	`viMoveIn8(ViSession vi, ViUInt16 space, ViBusAddress offset,ViBusSize length, ViAUInt8 buf8);`
Move 16-bit Value from Device Memory to Local Memory	`viMoveIn16(ViSession vi, ViUInt16 space, ViBusAddress offset,ViBusSize length, ViAUInt16 buf16);`
Move 32-bit Value from Device Memory to Local Memory	`viMoveIn32(ViSession vi, ViUInt16 space, ViBusAddress offset, ViBusSize length, ViAUInt32 buf32);`
Move 8-bit Value from Local Memory to Device Memory	`viMoveOut8(ViSession vi, ViUInt16 space, ViBusAddress offset, ViBusSize length, ViAUInt8 buf8);`

Operation	Function (Type *Parameter1*, Type *Parameter2*, ...);
Memory I/O (continued)	
Move 16-bit Value from Local Memory to Device Memory	`viMoveOut16(ViSession vi, ViUInt16 space, ViBusAddress offset, ViBusSize length, ViAUInt16 buf16);`
Move 32-bit Value from Local Memory to Device Memory	`viMoveOut32(ViSession vi, ViUInt16 space, ViBusAddress offset, ViBusSize length, ViAUInt32 buf32);`
Map Memory Space	`viMapAddress(ViSession vi, ViUInt16 mapSpace, ViBusAddress mapBase, ViBusSize mapSize, ViBoolean access, ViAddr suggested, ViAddr address);`
Unmap Memory Space	`viUnmapAddress(ViSession vi);`
Read 8-bit Value from Address	`viPeek8(ViSession vi, ViAddr addr, ViUInt8 val8);`
Read 16-bit Value from Address	`viPeek16(ViSession vi, ViAddr addr, ViUInt16 val16);`
Read 32-bit Value from Address	`viPeek32(ViSession vi, ViAddr addr, ViUInt32 val32);`
Write 8-bit Value to Address	`viPoke8(ViSession vi, ViAddr addr, ViUInt8 val8);`
Write 16-bit Value to Address	`viPoke16(ViSession vi, ViAddr addr, ViUInt16 val16);`
Write 32-bit Value to Address	`viPoke32(ViSession vi, ViAddr addr, ViUInt32 val32);`
GPIB Specific Services	
Control GPIB REN Interface Line	`viGpibControlREN(ViSession vi, ViUInt16 mode);`
Control GPIB ATN Interface Line	`viGpibControlATN(ViSession vi, ViUInt16 mode);`
Write GPIB Command Bytes on the bus	`viGpibCommand(ViSession vi, ViBuf buf, ViUInt32 count, ViUInt32 retCount );`
Tell GPIB Device to Become Controller in Charge (CIC)	`viGpibPassControl(ViSession vi, ViUInt16 primAddr, ViUInt16 secAddr);`
Pulse Interface Clear (IFC) Line	`viGpibSendIFC(ViSession vi);`

viAssertIntrSignal

Syntax

```
viAssertIntrSignal(ViSession vi, ViInt16 mode,
   ViUInt32 statusID);
```

Description

Asserts the specified device interrupt or signal. This operation can be used to assert a device interrupt condition. In VXI, for example, this can be done with either a VXI signal or a VXI interrupt. On certain bus types, the *statusID* parameter may be ignored.

> **NOTE**
>
> This function is *not* implemented in Agilent VISA.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
mode	IN	`ViInt16`	This specifies how to assert the interrupt. See the Description section for actual values.
statusID	IN	`ViUInte32`	This is the status value to be presented during an interrupt acknowledge cycle.

Special Values for *mode* Parameter

mode	Action Description
`VI_ASSERT_USE_ASSIGNED`	Use whatever notification method that has been assigned to the local device.
`VI_ASSERT_SIGNAL`	Send the notification via a VXI signal.

mode	Action Description
VI_ASSERT_IRQ1 - VI_ASSERT_IRQ7	Send the interrupt via the specified VXI/VME IRQ line. This uses the standard VXI/VME ROAK (release on acknowledge) interrupt mechanism rather than the older VME RORA (release on register access) mechanism.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INTR_PENDING	An interrupt is still pending from a previous call.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_NSUP_INTR	The interface cannot generate an interrupt on the requested level or with the requested *statusID* value.
VI_ERROR_NSUP_MODE	The specified *mode* is not supported by this VISA implementation.

See Also

BACKPLANE and SERVANT Resource Descriptions

viAssertTrigger

Syntax

viAssertTrigger(ViSession *vi*, **ViUInt16** *protocol*);

NOTE

This function is *not* supported with the GPIB-VXI interface.

Description

Assert software or hardware trigger. This operation will source a software or hardware trigger dependent on the interface type. For a GPIB device, the device is addressed to listen and then the GPIB *GET* command is sent.

For a VXI device, if **VI_ATTR_TRIG_ID** is **VI_TRIG_SW**, the device is sent the Word Serial *Trigger* command. For any other values of the attribute, a hardware trigger is sent on the line corresponding to the value of that attribute. For a GPIB device, if **VI_ATTR_TRIG_ID** is **VI_TRIG_SW**, the device is addressed to Listen and a Group Execute Trigger (GET) is sent.

For a serial session to a Serial device or TCPIP socket, if **VI_ATTR_IO_PROT** is **VI_PROT_4882_STRS**, the device is sent the string "*TRG\n". Otherwise, this operation is not valid.

In the Parameters table, the *protocol* values are:

- **VI_TRIG_PROT_DEFAULT** is **VI_TRIG_PROT_SYNC** for VXI
- **VI_TRIG_PROT_ON** asserts the trigger
- **VI_TRIG_PROT_OFF** deasserts the trigger
- **VI_TRIG_PROT_SYNC** pulses the trigger (assert followed by deassert)

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
protocol	IN	**ViUInt16**	Trigger protocol to use during assertion. Valid values are: **VI_TRIG_PROT_DEFAULT**, **VI_TRIG_PROT_ON**, **VI_TRIG_PROT_OFF**, and **VI_TRIG_PROT_SYNC**.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The specified trigger was successfully asserted to the device.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_PROT	The protocol specified is invalid.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_LINE_IN_USE	The specified trigger line is currently in use.
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).

Error Codes	Description
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

See Also

VI_ATTR_TRIG_ID attribute. Set this attribute to the trigger mechanism/ trigger line to use. **VI_EVENT_TRIGGER** description for details on trigger specifiers.

viAssertUtilSignal

Syntax

```
viAssertUtilSignal (ViSession vi, ViUInt16 line);
```

Description

Asserts the specified utility bus signal. This operation can be used to assert either the SYSFAIL or SYSRESET utility bus interrupts on the VXIbus backplane. This operation is valid only on VXI Mainframe Backplane (BACKPLANE) and on Servant Device-Side (SERVANT) resource sessions.

NOTE

This function is *not* supported in Agilent VISA.

Asserting SYSRESET (also known as HARD RESET in the VXI specification) should be used only when it is necessary to promptly terminate operation of all devices in a VXIbus system. This is a serious action that always affects the entire VXIbus system.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
line	IN	`ViUInt16`	Specifies the utility bus signal to assert. This can be the value `VI_UTIL_ASSERT_SYSRESET`, `VI_UTIL_ASSERT_SYSFAIL`, or `VI_UTIL_DEASSERT_SYSFAIL`

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_LINE	The value specified by the line parameter is invalid.

See Also

BACKPLANE and SERVANT Resource Descriptions

viBufRead

Syntax

viBufRead (ViSession *vi*, **ViBuf** *buf*, **ViUInt32** *count*, **ViUInt32** *retCount*) ;

Description

Similar to **viRead**, except that the operation uses the formatted I/O read buffer for holding data read from the device. This operation is similar to **viRead** and does not perform any kind of data formatting. It differs from **viRead** in that the data is read from the formatted I/O read buffer (the same buffer as used by **viScanf** and related operations) rather than directly from the device. This operation can intermix with the **viScanf** operation, but use with the **viRead** operation is discouraged.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	OUT	**ViBuf**	Represents the location of a buffer to receive data from the device.
count	IN	**ViUInt32**	Number of bytes to be read.
retCount	OUT	**ViUInt32**	Represents the location of an integer that will be set to the number of bytes actually transferred.

Special Values for *retCount* **Parameter**

Value	Action Description
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
`VI_SUCCESS_TERM_CHAR`	The specified termination character was read.
`VI_SUCCESS_MAX_CNT`	The number of bytes read is equal to *count*.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this function.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before function completed.
`VI_ERROR_IO`	An unknown I/O error occurred during transfer.

See Also

viWrite, **`viScanf`**

viBufWrite

Syntax

viBufWrite (ViSession *vi*, **ViBuf** *buf*, **ViUInt32** *count*, **ViUInt32** *retCount*);

Description

Similar to **viWrite**, except the data is written to the formatted I/O write buffer rather than directly to the device. This operation is similar to **viWrite** and does not perform any kind of data formatting.

It differs from **viWrite** in that the data is written to the formatted I/O write buffer (the same buffer as used by **viPrintf** and related operations) rather than directly to the device. This operation can intermix with the **viPrintf** operation, but mixing it with the **viWrite** operation is discouraged.

If you pass **VI_NULL** as the retCount parameter to the **viBufWrite** operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Represents the location of a data block to be sent to the device.
count	IN	**ViUInt32**	Number of bytes to be written.
retCount	OUT	**ViUInt32**	Represents the location of an integer that will be set to the number of bytes actually transferred.

Special Values for *retCount* Parameter

Value	Action Description
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

See Also

viWrite, **viBufRead**

viClear

Syntax

```
viClear(ViSession vi);
```

Description

Clear a device. This operation performs an IEEE 488.1-style clear of the device. For VXI, the Word Serial Clear command should be used. For GPIB systems, the Selected Device Clear command should be used. For a session to a Serial device or TCPIP socket, if **VI_ATTR_IO_PROT** is **VI_PROT_4882_STRS**, the device is sent the string "*CLS\n". Otherwise, this operation is not valid.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.

Error Codes	Description
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

viClose

Syntax

```
viClose(ViSession/ViEvent/ViFindList vi);
```

Description

This function closes the specified resource manager session, device session, find list (returned from the **viFindRsrc** function), or event context (returned from the **viWaitOnEvent** function, or passed to an event handler). In this process, all the data structures that had been allocated for the specified *vi* are freed.

NOTE

The **viClose** function should not be called from within an event handler. In VISA 1.1 and greater, **viClose** (**VI_NULL**) returns **VI_WARN_NULL_OBJECT** rather than an error.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession** **ViEvent** **ViFindList**	Unique logical identifier to a session, event, or find list.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Session closed successfully.
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
`VI_ERROR_CLOSING_FAILED`	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

See Also **viOpen, viFindRsrc, viWaitOnEvent, viEventHandler**

viDisableEvent

Syntax

```
viDisableEvent(ViSession vi, ViEventType eventType,
  ViUInt16 mechanism);
```

Description

This function disables servicing of an event identified by the *eventType* parameter for the mechanisms specified in the *mechanism* parameter. Specifying **VI_ALL_ENABLED_EVENTS** for the *eventType* parameter allows a session to stop receiving all events.

The session can stop receiving queued events by specifying **VI_QUEUE**. Applications can stop receiving callback events by specifying either **VI_HNDLR** or **VI_SUSPEND_HNDLR**. Specifying **VI_ALL_MECH** disables both the queuing and callback mechanisms.

viDisableEvent prevents new event occurrences from being added to the queue(s). However, event occurrences already existing in the queue(s) are not discarded.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
eventType	IN	**ViEventType**	Logical event identifier. (See the following tables.)
mechanism	IN	**ViUInt16**	Specifies event handling mechanisms to be disabled. The queuing mechanism is disabled by specifying **VI_QUEUE.** The callback mechanism is disabled by specifying **VI_HNDLR** or **VI_SUSPEND_HNDLR**. It is possible to disable both mechanisms simultaneously by specifying **VI_ALL_MECH**.

Special Values for *eventType* Parameter

Value	Action Description
`VI_ALL_ENABLED_EVENTS`	Disable all events that were previously enabled.

The following events can be disabled:

Event Name	Description
`VI_EVENT_SERVICE_REQ`	Notification that a device is requesting service.
`VI_EVENT_VXI_SIGP`	Notification that a VXI signal or VXI interrupt has been received from a device.
`VI_EVENT_TRIG`	Notification that a hardware trigger was received from a device.
`VI_EVENT_IO_COMPLETION`	Notification that an asynchronous operation has completed.

Special Values for *mechanism* Parameter

Value	Action Description
`VI_QUEUE`	Disable this session from receiving the specified event(s) via the waiting queue.
`VI_HNDLR` or `VI_SUSPEND_HNDLR`	Disable this session from receiving the specified event(s) via a callback handler or a callback queue.
`VI_ALL_MECH`	Disable this session from receiving the specified event(s) via any mechanism.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Event disabled successfully.
`VI_SUCCESS_EVENT_DIS`	Specified event is already disabled for at least one of the specified mechanisms.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

See Also

See the handler prototype **viEventHandler** for its parameter description, and **viEnableEvent**. Also, see **viInstallHandler** and **viUninstallHandler** descriptions for information about installing and uninstalling event handlers. See event descriptions for context structure definitions.

viDiscardEvents

Syntax

```
viDiscardEvents(ViSession vi, ViEventType eventType,
  ViUInt16 mechanism);
```

Description

This function discards all pending occurrences of the specified event types for the mechanisms specified in a given session. The information about all the event occurrences which have not yet been handled is discarded. This function is useful to remove event occurrences that an application no longer needs.

The event occurrences discarded by applications are not available to a session at a later time. This operation causes loss of event occurrences. The **viDiscardEvents** operation does not apply to event contexts that have already been delivered to the application.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
eventType	IN	**ViEventType**	Logical event identifier. (See the following tables.)
mechanism	IN	**ViUInt16**	Specifies the mechanisms for which the events are to be discarded. **VI_QUEUE** is specified for the queuing mechanism and **VI_SUSPEND_HNDLR** is specified for the pending events in the callback mechanism. It is possible to specify both mechanisms simultaneously by specifying **VI_ALL_MECH**.

Special Values for *eventType* Parameter

Value	Action Description
`VI_ALL_ENABLED_EVENTS`	Discard events of every type that is enabled.

The following events can be discarded**:**

Event Name	Description
`VI_EVENT_SERVICE_REQ`	Notification that a device is requesting service.
`VI_EVENT_VXI_SIGP`	Notification that a VXI signal or VXI interrupt has been received from a device.
`VI_EVENT_TRIG`	Notification that a hardware trigger was received from a device.
`VI_EVENT_IO_COMPLETION`	Notification that an asynchronous operation has completed.

Special Values for *mechanism* Parameter

Value	Action Description
`VI_QUEUE`	Discard the specified event(s) from the waiting queue.
`VI_SUSPEND_HNDLR`	Discard the specified event(s) from the callback queue.
`VI_ALL_MECH`	Discard the specified event(s) from all mechanisms.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Event queue flushed successfully.
VI_SUCCESS_QUEUE_EMPTY	Operation completed successfully, but queue was empty.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

See Also

viEnableEvent, viWaitOnEvent, viInstallHandler

viEnableEvent

Syntax

```
viEnableEvent(ViSession vi, ViEventType eventType,
  ViUInt16 mechanism, ViEventFilter context);
```

Description

This function enables notification of an event identified by the *eventType* parameter for mechanisms specified in the *mechanism* parameter. The specified session can be enabled to queue events by specifying **VI_QUEUE**.

NOTE

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

Applications can enable the session to invoke a callback function to execute the handler by specifying **VI_HNDLR**. The applications are required to install at least one handler to be enabled for this mode.

Specifying **VI_SUSPEND_HNDLR** enables the session to receive callbacks, but the invocation of the handler is deferred to a later time. Successive calls to this function replace the old callback mechanism with the new callback mechanism.

Specifying **VI_ALL_ENABLED_EVENTS** for the *eventType* parameter refers to all events which have previously been enabled on this session, making it easier to switch between the two callback mechanisms for multiple events.

Event queuing and callback mechanisms operate completely independently. As such, enabling and disabling of the two modes in done independently (enabling one of the modes does not enable or disable the other mode). For example, if **viEnableEvent** is called once with **VI_HNDLR** and called a second time with **VI_QUEUE**, both modes would be enabled.

If **viEnableEvent** is called with the *mechanism* parameter equal to the "bit-wise OR" of **VI_SUSPEND_HNDLR** and **VI_HNDLR**, **viEnableEvent** returns **VI_ERROR_INV_MECH**.

If the event handling mode is switched from **VI_SUSPEND_HNDLR** to **VI_HNDLR** for an event type, handlers that are installed for the event are called once for each occurrence of the corresponding event pending in the session (and dequeued from the suspend handler queue) before switching the modes.

A session enabled to receive events can start receiving events before the **viEnableEvent** operation returns. In this case, the handlers set for an event type are executed before the completion of the enable operation.

If the event handling mode is switched from **VI_HNDLR** to **VI_SUSPEND_HNDLR** for an event type, handler invocation for occurrences of the event type is deferred to a later time. If no handler is installed for an event type, the request to enable the callback mechanism for the event type returns **VI_ERROR_HNDLR_NINSTALLED**.

If a session has events pending in its queue(s) and **viClose** is invoked on that session, all pending event occurrences and the associated event contexts that have not yet been delivered to the application for that session are freed by the system.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
eventType	IN	**ViEventType**	Logical event identifier.
mechanism	IN	**ViUInt16**	Specifies event handling mechanisms to be enabled. The queuing mechanism is enabled by **VI_QUEUE**, and the callback mechanism is enabled by **VI_HNDLR** or **VI_SUSPEND_HNDLR**. It is possible to enable both mechanisms simultaneously by specifying "bit-wise OR" of **VI_QUEUE** and one of the two mode values for the callback mechanism.

Name	Direction	Type	Description
context	IN	**ViEventFilter**	**VI_NULL** (Not used for VISA 1.0.)

Special Values for *eventType* Parameter

Value	Action Description
VI_ALL_ENABLED_EVENTS	Switch all events that were previously enabled to the callback mechanism specified in the mechanism parameter.

The following events can be enabled:

Event Name	Description
VI_EVENT_SERVICE_REQ	Notification that a device is requesting service.
VI_EVENT_VXI_SIGP	Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_TRIG	Notification that a hardware trigger was received from a device.
VI_EVENT_IO_COMPLETION	Notification that an asynchronous operation has completed.

Special Values for *mechanism* Parameter

Value	Action Description
VI_QUEUE	Enable this session to receive the specified event via the waiting queue. Events must be retrieved manually via the **viWaitOnEvent** function.
VI_HNDLR	Enable this session to receive the specified event via a callback handler, which must have already been installed via **viInstallHandler**.
VI_SUSPEND_HNDLR	Enable this session to receive the specified event via a callback queue. Events will not be delivered to the session until **viEnableEvent** is invoked again with the **VI_HNDLR** mechanism.

NOTE

Any combination of VISA-defined values for different parameters of this function is also supported (except for **VI_HNDLR** and **VI_SUSPEND_HNDLR**, which apply to different modes of the same mechanism).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Event enabled successfully.
VI_SUCCESS_EVENT_EN	Specified event is already enabled for at least one of the specified mechanisms.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.
VI_ERROR_INV_CONTEXT	Specified event context is invalid.
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event. The session cannot be enabled for the **VI_HNDLR** mode of the callback mechanism.
VI_ERROR_NSUP_MECH	The specified mechanism is not supported for the given event type.

See Also

See the handler prototype **`viEventHandler`** for its parameter description and **`viDisableEvent`**. Also, see the **`viInstallHandler`** and **`viUninstallHandler`** descriptions for information about installing and uninstalling event handlers.

viEventHandler

Syntax

```
viEventHandler(ViSession vi, ViEventType eventType,
  ViEvent context, ViAddr userHandle);
```

Description

This is a prototype for a function, which you define. The function you define is called whenever a session receives an event and is enabled for handling events in the **VI_HNDLR** mode. The handler services the event and returns **VI_SUCCESS** on completion. VISA event handlers must be declared as follows.

```
ViStatus _VI_FUNCH MyEventHandler(ViSession vi,
  ViEventType eventType, ViEvent context,
  ViAddr userHandle);
```

The **_VI_FUNCH** declaration is required to make sure the handler is of the proper type. If **_VI_FUNCH** is not included, stack corruption may occur on the function call or return. The **_VI_FUNCH** declaration is very important since it declares the function of type *stdcall* which VISA requires. Visual Studio C++ defaults to *cdecl* which will not work. When the handler returns, it will generate an access violation because the stack gets corrupted.

Because each *eventType* defines its own context in terms of attributes, refer to the appropriate event definition to determine which attributes can be retrieved using the *context* parameter.

Because the event context must still be valid after the user handler returns (so that VISA can free it up), an application should not invoke the **viClose** operation on an event context passed to a user handler.

If the user handler will not return to VISA, the application should call **viClose** on the event context to manually delete the event object. This may occur when a handler throws a C++ exception in response to a VISA exception event.

Normally, an application should return **VI_SUCCESS** from all callback handlers. If a specific handler does not want other handlers to be invoked for the given event for the given session, it should return **VI_SUCCESS_NCHAIN**. No return value from a handler on one session will affect callbacks on other sessions.

NOTE

Future versions of VISA (or specific implementations of VISA) may take actions based on other return values, so users should return **VI_SUCCESS** from handlers unless there is a specific reason to do otherwise.

This table lists events and associated read-only attributes implemented by Agilent VISA that can be read to get event information on a specific event. Use the **viReadSTB** function to read the status byte of the service request.

Instrument Control (INSTR) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_SERVICE_REQUEST	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_REQ
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_STOP
	VI_ATTR_SIGP_STATUS_ID	ViUInt16	0 to FFFFh
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIG
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

Memory Access (MEMACC) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_IO_ COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_ COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

GPIB Bus Interface (INTFC) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_GPIB_CIC	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_CIC
	VI_ATTR_GPIB_RECV_CIC_ STATE	ViBoolean	VI_TRUE VI_FALSE
VI_EVENT_GPIB_TALK	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_TALK
VI_EVENT_GPIB_ LISTEN	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_ LISTEN
VI_EVENT_CLEAR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_CLEAR
VI_EVENT_TRIGGER	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIGGER
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_SW
VI_EVENT_IO_ COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_ COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

VXI Mainframe Backplane (BACKPLANE) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIG
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_EVENT_VXI_VME_SYSFAIL	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSFAIL
VI_EVENT_VXI_VME_SYSRESET	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSRESET

TCPIP Socket (SOCKET) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
eventType	IN	ViEventType	Logical event identifier.
context	IN	ViEvent	A handle specifying the unique occurrence of an event.
userHandle	IN	ViAddr	A value specified by an application that can be used for identifying handlers uniquely in a session for an event.

Return Values

Type **`ViStatus`** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Event enabled successfully.

See Also

See *Chapter 4 - Programming with VISA* for more information on event handling and exception handling.

viFindNext

Syntax

viFindNext(ViFindList *findList*, **ViPRsrc** *instrDesc*);

Description

This function returns the next resource found in the list created by **viFindRsrc**. The list is referenced by the handle that was returned by **viFindRsrc**.

Parameters

Name	Direction	Type	Description
findList	IN	**ViFindList**	Describes a find list. This parameter must be created by **viFindRsrc**.
instrDesc	OUT	**ViPRsrc**	Returns a string identifying location of a device. Strings can be passed to **viOpen** to establish a session to the device.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Resource(s) found.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	Given *findList* does not support this function.
VI_ERROR_RSRC_NFOUND	There are no more matches.

See Also

viFindRsrc

viFindRsrc

Syntax

```
viFindRsrc(ViSession sesn, ViString expr, ViFindList
   findList, ViUInt32 retcnt, ViRsrc instrDesc);
```

Description

This function queries a VISA system to locate the resources associated with a specified interface. This function matches the value specified in the *expr* parameter with the resources available for a particular interface.

On successful completion, it returns the first resource found in the list and returns a count to indicate if there were more resources found that match the value specified in the *expr* parameter.

This function also returns a handle to a find list. This handle points to the list of resources, and it must be used as an input to **viFindNext**. When this handle is no longer needed, it should be passed to **viClose**.

The search criteria specified in the *expr* parameter has two parts: a regular expression over a resource string and an optional logical expression over attribute values. The regular expression is matched against the resource strings of resources known to the VISA Resource Manager.

If the resource string matches the regular expression, the attribute values of the resource are then matched against the expression over attribute values. If the match is successful, the resource has met the search criteria and gets added to the list of resources found. (Agilent VISA does not support matching of attribute values.)

The optional attribute expression allows construction of expressions with the use of logical ANDs, ORs and NOTs. Equal (==) and unequal (!=) comparators can be used compare attributes of any type. In addition, other inequality comparators (>, <, >=, <=) can be used to compare attributes of numeric type. Only global attributes can be used in the attribute expression.

The syntax of *expr* is defined as follows. The grouping operator `()` in a logical expression has the highest precedence, The not operator `!` in a logical expression has the next highest precedence after the grouping operator, and the or operator `||` in a logical expression has the lowest precedence. (Agilent VISA does not support the use of logical expressions over all attribute values.)

Special Character	Meaning
&&	Logical AND
||	Logical OR
!	Logical negation (NOT)
()	Parentheses

```
expr :=
            regularExpr ['{' attrExpr '}']
attrExpr :=
            attrTerm |
            attrExpr '||' attrTerm
attrTerm :=
            attrFactor |
            attrTerm '&&' attrFactor
attrFactor :=
            '(' attrExpr ')' |
            '!' attrFactor  |
            relationExpr
 relationExpr :=
            attributeId compareOp  numValue |
            attributeId equalityOp stringValue
compareOp :=
            '==' | '!=' | '>' | '<' | '>=' | '<='
equalityOp :=
            '==' | '!='
attributeId :=
            character (character|digit|underscore)*
numValue :=
            digit+ |
            '-' digit+ |
            '0x' hex_digit+ |
            '0X' hex_digit+
stringValue :=
            '"' character* '"'
```

Some examples are:

Expr	Meaning
`GPIB[0-9]*::?*::?*::INSTR {VI_ATTR_GPIB_SECONDARY_ADDR > 0}`	Find all GPIB devices that have secondary addresses greater than 0.
`ASRL?*INSTR{VI_ATTR_ASRL_BAUD == 9600}`	Find all serial ports configured at 9600 baud.
`?*VXI?*INSTR{VI_ATTR_MANF_ID == 0xFF6 && !(VI_ATTR_VXI_LA == 0 || VI_ATTR_SLOT <= 0)}`	Find all VXI instrument resources whose manufacturer ID is FF6 and who are not logical address 0, slot 0, or external controllers.

Local attributes are not allowed in the logical expression part of the *expr* parameter to the **viFindRsrc** operation. **viFindRsrc** uses a case-insensitive compare function when matching resource names against the regular expression specified in *expr*.

If the value **VI_NULL** is specified in the *findList* parameter of **viFindRsrc** and the return value is successful, VISA automatically invokes **viClose** on the find list handle rather than returning it to the application.

The *findList* and *retCnt* parameters to the **viFindRsrc** operation are optional. They can be used if only the first match is important and the number of matches is not needed. Calling **viFindRsrc** with **"VXI?*INSTR"** will return the same resources as invoking it with **"vxi?*instr"**.

All resource strings returned by **viFindRsrc** must be recognized by **viParseRsrc** and **viOpen**. However, not all resource strings that can be parsed or opened have to be findable.

Parameters

Name	Direction	Type	Description
sesn	IN	**ViSession**	Resource Manager session (should always be the Default Resource Manager for VISA returned from **viOpenDefaultRM**).
expr	IN	**ViString**	This expression sets the criteria to search an interface or all interfaces for existing devices. (See the following table for description string format.)
findList	OUT	**ViFindList**	Returns a handle identifying this search session. This handle will be used as an input in **viFindNext**.
retcnt	OUT	**ViUInt32**	Number of matches.
instrDesc	OUT	**ViRsrc**	Returns a string identifying the location of a device. Strings can then be passed to **viOpen** to establish a session to the given device.

Description String for *expr* Parameter

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

Special Values for *findList* Parameter

Value	Action Description
VI_NULL	Do not return a find list handle.

Special Values for *retcnt* Parameter

Value	Action Description
`VI_NULL`	Do not return the number of matches.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Resource(s) found.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *sesn* does not support this function.
`VI_ERROR_INV_EXPR`	Invalid expression specified for search.
`VI_ERROR_RSRC_NFOUND`	Specified expression does not match any devices.

See Also

viFindNext, viClose

viFlush

Syntax

```
viFlush(ViSession vi, ViUInt16 mask);
```

Description

Manually flush the specified buffers associated with formatted I/O operations and/or serial communication. The values for *the mask* parameter are:

Flag	Interpretation
VI_READ_BUF	Discard the read buffer contents and, if data was present in the read buffer and no END-indicator was present, read from the device until encountering an END indicator (which causes the loss of data). This action resynchronizes the next **viScanf** call to read a <TERMINATED RESPONSE MESSAGE>. (See the IEEE 488.2 standard.)
VI_READ_BUF_DISCARD	Discard read buffer contents (does not perform any I/O to the device).
VI_WRITE_BUF	Flush the write buffer by writing all buffered data to the device.
VI_WRITE_BUF_DISCARD	Discard write buffer contents (does not perform any I/O to the device).
VI_IO_IN_BUF	Discard receive buffer contents (same as **VI_IO_IN_BUF_DISCARD**).
VI_IO_IN_BUF_DISCARD	Discard eceive buffer contents (does not perform an I/O to the device).
VI_IO_OUT_BUF	Flush the transmit buffer by writing all buffered data to the device.
VI_IO_OUT_BUF_DISCARD	Discard transmit buffer contents (does not perform any I/O to the device).

It is possible to combine any of these read flags and write flags for different buffers by ORing the flags. However, combining two flags for the same buffer in the same call to **viFlush** is illegal.

When using formatted I/O operations with a serial device, a flush of the formatted I/O buffers also causes the corresponding serial communication buffers to be flushed. For example, calling **viFlush** with **VI_WRITE_BUF** also flushes the **VI_IO_OUT_BUF**.

For backward compatibility, **VI_IO_IN_BUF** is the same as **VI_ASRL_IN_BUF**, **VI_IO_IN_BUF_DISCARD** is the same as **VI_ASRL_IN_BUF_DISCARD**, **VI_IO_OUT_BUF** is the same as **VI_ASRL_OUT_BUF**, and **VI_IO_OUT_BUF_DISCARD** is the same as **VI_ASRL_OUT_BUF_DISCARD**.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
mask	IN	`ViUInt16`	Specifies the action to be taken with flushing the buffer. (See the following table.)

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Buffers flushed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read/write function because of I/O error.
`VI_ERROR_TMO`	The read/write function was aborted because timeout expired while function was in progress.
`VI_ERROR_INV_MASK`	The specified *mask* does not specify a valid flush function on read/write resource.

See Also

viSetBuf

viGetAttribute

Syntax

```
viGetAttribute(ViSession/ViEvent/ViFindList vi,
  ViAttr attribute, ViAttrState attrState);
```

Description

This function retrieves the state of an attribute for the specified session.

Parameters

Name	Direction	Type	Description

vi	IN	`ViSession`
		`ViEvent`
		`ViFindList`
			Unique logical identifier to a session, event, or find list.

attribute	IN	`ViAttr`	Resource attribute for which the state query is made.
attrState	OUT	See Note below.	The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource. Note that you must allocate space for character strings returned.

NOTE

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 **`ViChar`**, you must pass a pointer to a **`ViChar`** variable. You must allocate space for the returned data.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Resource attribute retrieved successfully.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced resource.

See Also

viSetAttribute

viGpibCommand

Syntax

```
viGpibCommand(ViSession vi, ViBuf buf, ViUInt32 count,
   ViUInt32 retCount);
```

Description

Write GPIB command bytes on the bus. This operation attempts to write count number of bytes of GPIB commands to the interface bus specified by *vi*. This operation is valid only on GPIB INTFC (interface) sessions. This operation returns only when the transfer terminates.

If you pass **VI_NULL** as the *retCount* parameter to the **viGpibCommand** operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Buffer containing valid GPIB commands.
count	IN	**ViUInt32**	Number of bytes to be written.
retCount	IN	**ViUInt32**	Number of bytes actually transferred.

Special Value for *retCount* Parameter

Value	Action Description
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Resource attribute retrieved successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

See Also

INTFC Resource Description

viGpibControlATN

Syntax

```
viGpibControlATN(ViSession vi, ViUInt16 mode);
```

Description

Controls the state of the GPIB ATN interface line, and optionally the active controller state of the local interface board. This operation asserts or deasserts the GPIB ATN interface line according to the specified mode. The mode can also specify whether the local interface board should acquire or release Controller Active status. This operation is valid only on GPIB INTFC (interface) sessions.

NOTE

It is generally not necessary to use the **viGpibControlATN** operation in most applications. Other operations such as **viGpibCommand** and **viGpibPassControl** modify the ATN and/or CIC state automatically.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
mode	IN	**ViUInt16**	Specifies the state of the ATN line and, optionally, the local active controller state.

Special Values for *mode* Parameter

mode	Action Description
VI_GPIB_ATN_DEASSERT	Deassert ATN line.
VI_GPIB_ATN_ASSERT	Assert ATN line synchronously (in 488 terminology). If a data handshake is in progress, ATN will not be asserted until the handshake is complete.
VI_GPIB_ATN_DEASSERT_HANDSHAKE*	Deassert ATN line, and enter shadow handshake mode. The local board will participate in data handshakes as an Acceptor without actually reading the data.

mode	Action Description
VI_GPIB_REN_ASSERT_ IMMEDIATE*	Assert ATN line asynchronously (in 488 terminology). This should generally be used only under error conditions.

* Not supported in Agilent VISA

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_NSUP_MODE	The specified mode is not supported by this VISA implementation.

See Also

INTFC Resource Description

viGpibControlREN

Syntax

```
viGpibControlREN(ViSession vi, ViUInt16 mode);
```

Description

Controls the state of the GPIB REN interface line and, optionally, the remote/local state of the device. This operation asserts or deasserts the GPIB REN interface line according to the specified mode.

The mode can also specify whether the device associated with this session should be placed in local state (before deasserting REN) or remote state (after asserting REN). This operation is valid only if the GPIB interface associated with the session specified by *vi* is currently the system controller.

An INSTR resource implementation of **viGpibControlREN** for a GPIB System supports all documented modes. An INTFC resource implementation of **viGpibControlREN** for a GPIB System supports the modes **VI_GPIB_REN_DEASSERT**, **VI_GPIB_REN_ASSERT**, and **VI_GPIB_REN_ASSERT_LLO**.

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
mode	IN	ViUInt16	Specifies the state of the REN line and, optionally, the device remote/local state.

Special Values for *mode* Parameter

mode	Action Description
VI_GPIB_REN_DEASSERT	Deassert REN line.
VI_GPIB_REN_ASSERT	Assert REN line.
VI_GPIB_REN_DEASSERT_GTL	Send the Go To Local command (GTL) to this device and deassert REN line.
VI_GPIB_REN_ASSERT_ADDRESS	Assert REN line and address this device.
VI_GPIB_REN_ASSERT_LLO	Send LLO to any devices that are addressed to listen.

mode	Action Description
`VI_GPIB_REN_ASSERT_ADDRESS_LLO`	Address this device and send it LLO, putting it in RWLS.
`VI_GPIB_REN_ADDRESS_GTL`	Send the Go To Local command (GTL) to this device.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Resource attribute retrieved successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_NCIC`	The interface associated with this session is not currently the controller in charge.
`VI_ERROR_NLISTENERS`	No listeners condition is detected (both NRFD and NDAC are deasserted).
`VI_ERROR_NSYS_CNTLR`	The interface associated with this session is not the system controller.
`VI_ERROR_INV_MODE`	The value specified by the *mode* parameter is invalid.

viGpibPassControl

Syntax

```
viGpibPassControl(ViSession vi, ViUInt16 primAddr,
  ViUInt16 secAddr);
```

Description

Tell the GPIB device at the specified address to become controller in charge (CIC). This operation passes controller in charge status to the device indicated by *primAddr* and *secAddr* and then deasserts the ATN line. This operation assumes that the targeted device has controller capability. This operation is valid only on GPIB INTFC (interface) sessions.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
primAddr	IN	`ViUInt16`	Primary address of the GPIB device to which you want to pass control.
secAddr	IN	`ViUInt16`	Secondary address of the targeted GPIB device. If the targeted device does not have a secondary address, this parameter should contain the value `VI_NO_SEC_ADDR`.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

See Also

INTFC Resource Description

viGpibSendIFC

Syntax

```
viGpibSendIFC(ViSession vi);
```

Description

Pulse the interface clear line (IFC) for at least 100 µseconds. This operation asserts the IFC line and becomes controller in charge (CIC). The local board must be the system controller. This operation is valid only on GPIB INTFC (interface) sessions.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_NSYS_CNTLR`	The interface associated with this session is not the system controller.

See Also

INTFC Resource Description

viIn8, viIn16, and viIn32

Syntax

```
viIn8(ViSession vi, ViUInt16 space, ViBusAddress offset,
   ViUInt8 val8);

viIn16(ViSession vi, ViUInt16 space, ViBusAddress offset,
   ViUInt16 val16);

viIn32(ViSession vi, ViUInt16 space, ViBusAddress offset,
   ViUInt32 val32);
```

Description

This operation, by using the specified address space, reads in 8, 16, or 32 bits of data from the specified offset. This operation does not require **viMapAddress** to be called prior to its invocation.

This function reads in an 8-bit, 16-bit, or 32-bit value from the specified memory space (assigned memory base + *offset*). This function takes the 8-bit, 16-bit, or 32-bit value from the address space pointed to by *space*. The *offset* must be a valid memory address in the *space*. This function does not require **viMapAddress** to be called prior to its invocation.

If the **viSession** parameter (*vi*) refers to an INSTR session, the *offset* parameter specifies a relative offset from the start of the instrument's address *space*. If the **viSession** parameter (*vi*) refers to a MEMACC session, the *offset* parameter is an absolute offset from the start of memory in that VXI address *space*. The valid entries for specifying address space are:

Value	Description
VI_A16_SPACE	Address the A16 address space of VXI/MXI bus.
VI_A24_SPACE	Address the A24 address space of VXI/MXI bus.
VI_A32_SPACE	Address the A32 address space of VXI/MXI bus.

The high-level operations **viIn8**, **viIn16**, and **viIn32** operate successfully independently from the low-level operations (**viMapAddress**, **viPeek8, viPeek16, viPeek32, viPoke8, viPoke16**, and **viPoke32**). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

For an INSTR resource, the *offset* is a relative address of the device associated with the given INSTR resource. For a MEMACC resource, the *offset* parameter specifies an absolute address.

The *offset* specified in the **viIn8, viIn16**, and **viIn32** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified.

For example, if *space* specifies **VI_A16_SPACE**, *offset* specifies the offset from the logical address base address of the VXI device specified. If *space* specifies **VI_A24_SPACE** or **VI_A32_SPACE**, *offset* specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
space	IN	**ViUInt16**	Specifies the address space. (See the following table.)
offset	IN	**ViBusAddress**	Offset (in bytes) of the memory to read from.
val8, val16, or val32	OUT	**ViUInt8, ViUInt16,** or **ViUInt32**	Data read from bus (8-bits for **viIn8**, 16-bits for **viIn16**, and 32-bits for **viIn32**).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.

See Also

viOut8, viOut16, viOut32, viPeek8, viPeek16, viPeek32, viMoveIn8, viMoveIn16, viMoveIn32

viInstallHandler

Syntax

```
viInstallHandler(ViSession vi, ViEventType eventType,
  ViHndlr handler, ViAddr userHandle);
```

Description

This function allows applications to install handlers on sessions for event callbacks. The handler specified in the *handler* parameter is installed along with previously installed handlers for the specified event. Applications can specify a value in the *userHandle* parameter that is passed to the handler on its invocation. VISA identifies handlers uniquely using the handler reference and the *userHandle* value.

NOTE

Versions of VISA prior to Version 2.0 allow only a single handler per event type per session.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
eventType	IN	`ViEventType`	Logical event identifier.
handler	IN	`ViHndlr`	Interpreted as a valid reference to a handler to be installed by an application.
userHandle	IN	`ViAddr`	A value specified by an application that can be used for identifying handlers uniquely for an event type.

The following events can be enabled:

Event Name	Description
`VI_EVENT_SERVICE_REQ`	Notification that a device is requesting service.
`VI_EVENT_VXI_SIGP`	Notification that a VXI signal or VXI interrupt has been received from a device.

Event Name	Description
`VI_EVENT_TRIG`	Notification that a hardware trigger was received from a device.
`VI_EVENT_IO_COMPLETION`	Notification that an asynchronous operation has completed.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Event handler installed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_INV_EVENT`	Specified event type is not supported by the resource.
`VI_ERROR_INV_HNDLR_REF`	The given handler reference is invalid.
`VI_ERROR_HNDLR_NINSTALLED`	The handler was not installed. This may be returned if an application attempts to install multiple handlers for the same event on the same session.

See Also

viEventHandler

viLock

Syntax

viLock(ViSession *vi*, **ViAccessMode** *lockType*, **ViUInt32** *timeout*, **ViKeyId** *requestedKey*, **ViKeyId** *accessKey*);

NOTE

The **viLock** function is *not* supported on network devices.

Description

This function is used to obtain a lock on the specified resource. The caller can specify the type of lock requested (exclusive or shared lock) and the length of time the operation will suspend while waiting to acquire the lock before timing out. This function can also be used for sharing and nesting locks.

The *requestedKey* and *accessKey* parameters apply only to shared locks. These parameters are not applicable when using the lock type **VI_EXCLUSIVE_LOCK**. In this case, *requestedKey* and *accessKey* should be set to **VI_NULL**. VISA allows user applications to specify a key to be used for lock sharing through the use of the *requestedKey* parameter.

Alternatively, a user application can pass **VI_NULL** for the *requestedKey* parameter when obtaining a shared lock, in which case VISA will generate a unique access key and return it through the *accessKey* parameter. If a user application does specify a *requestedKey* value, VISA will try to use this value for the *accessKey*.

As long as the resource is not locked, VISA will use the *requestedKey* as the access key and grant the lock. When the operation succeeds, the *requestedKey* will be copied into the user buffer referred to by the *accessKey* parameter.

The session that gained a shared lock can pass the *accessKey* to other sessions for the purpose of sharing the lock. The session wanting to join the group of sessions sharing the lock can use the key as an input value to the *requestedKey* parameter.

VISA will add the session to the list of sessions sharing the lock, as long as the *requestedKey* value matches the *accessKey* value for the particular resource. The session obtaining a shared lock in this manner will then have the same access privileges as the original session that obtained the lock.

It is also possible to obtain nested locks through this function. To acquire nested locks, invoke the **viLock** function with the same lock type as the previous invocation of this function. For each session, **viLock** and **viUnlock** share a lock count, which is initialized to **0**. Each invocation of **viLock** for the same session (and for the same *lockType*) increases the lock count.

A shared lock returns with the same *accessKey* every time. When a session locks the resource a multiple number of times, it is necessary to invoke the **viUnlock** function an equal number of times in order to unlock the resource. That is, the lock count increments for each invocation of **viLock**, and decrements for each invocation of **viUnlock**. A resource is actually unlocked only when the lock count is **0**.

NOTE

On HP-UX, SIGALRM is used in implementing the **viLock** when *timeout* is non-zero. The **viLock** function's use of SIGALRM is exclusive – an application should not also expect to use SIGALRM at the same time.

NOTE

On HP-UX, some semaphores used in locking are permanently allocated and diminish the number of semaphores available for applications. If the operating system runs out of semaphores, the number of semaphores may be increased by doing the following:

1. Run **sam**.
2. Double-click **Kernel Configuration**.
3. Double-click **Configurable Parameters**.
4. Change **semmni** and **semmns** to a higher value, such as **300**.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
lockType	IN	**ViAccessMode**	Specifies the type of lock requested, which can be **VI_EXCLUSIVE_LOCK** or **VI_SHARED_LOCK**.

Name	Direction	Type	Description
timeout	IN	**ViUInt32**	Absolute time period (in milliseconds) that a resource waits to get unlocked by the locking session before returning this operation with an error. **VI_TMO_IMMEDIATE** and **VI_TMO_INFINITE** are also valid values.
requestedKey	IN	**ViKeyId**	This parameter is not used and should be set to **VI_NULL** when **lockType** is **VI_EXCLUSIVE_LOCK** (exclusive lock). When trying to lock the resource as **VI_SHARED_LOCK** (shared lock), a session can either set it to **VI_NULL** so that VISA generates an *accessKey* for the session, or the session can suggest an *accessKey* to use for the shared lock. See "Description" for more details.
accessKey	OUT	**ViKeyId**	This parameter should be set to **VI_NULL** when *lockType* is **VI_EXCLUSIVE_LOCK** (exclusive lock). When trying to lock the resource as **VI_SHARED_LOCK** (shared lock), the resource returns a unique access key for the lock if the operation succeeds. This *accessKey* can then be passed to other sessions to share the lock.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	The specified access mode was successfully acquired.
VI_SUCCESS_NESTED_EXCLUSIVE	The specified access mode was successfully acquired, and this session has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The specifed access mode was successfully acquired, and this session has nested shared locks.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given *vi* does not identify a valid session or object.
VI_ERROR_RSRC_LOCKED	The specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_ACCESS_KEY	The *requestedKey* value passed is not a valid access key to the specified resource.
VI_ERROR_TMO	The specified type of lock could not be obtained within the specified timeout period.

See Also

viUnlock. For more information on locking, see *Chapter 4 - Programming with VISA*.

viMapAddress

Syntax

```
viMapAddress(ViSession vi, ViUInt16 mapSpace,
  ViBusAddress mapBase, ViBusSize mapSize,
  ViBoolean access, ViAddr suggested,ViAddr address);
```

Description

This function maps in a specified memory space. The memory space that is mapped is dependent on the type of interface specified by the *vi* parameter and the *mapSpace* parameter (see the following table). The *address* parameter returns the address in your process space where memory is mapped. The values for the *mapSpace* parameter are:

Value	Description
VI_A16_SPACE	Map the A16 address space of VXI/MXI bus.
VI_A24_SPACE	Map the A24 address space of VXI/MXI bus.
VI_A32_SPACE	Map the A32 address space of VXI/MXI bus.

If the **viSession** parameter (*vi*) refers to an INSTR session, the *mapBase* parameter specifies a relative offset in the instrument's *mapSpace*. If the **viSession** parameter (*vi*) refers to a MEMACC session, the *mapBase* parameter is an absolute offset from the start of the VXI *mapSpace*.

NOTE

For a given session, you can only have one map at one time. If you need to have multiple maps to a device, you must open one session for each map needed.

The *mapBase* parameter specified in the **viMapAddress** operation for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified.

For example, if *mapSpace* specifies **VI_A16_SPACE**, *mapBase* specifies the offset from the logical address base address of the VXI device specified. If *mapSpace* specifies **VI_A24_SPACE** or **VI_A32_SPACE**, *mapBase* specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
mapSpace	IN	**ViUInt16**	Specifies the address space to map.
mapBase	IN	**ViBusAddress**	Offset (in bytes) of the memory to be mapped.
mapSize	IN	**ViBusSize**	Amount of memory to map (in bytes).
access	IN	**ViBoolean**	**VI_FALSE**.
suggested	IN	**ViAddr**	If suggested parameter is not **VI_NULL**, the operating system attempts to map the memory to the address specified in *suggested*. There is no guarantee, however, that the memory will be mapped to that address. This function may map the memory into an address region different from *suggested*.
address	OUT	**ViAddr**	Address in your process space where the memory was mapped.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Map successful.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given vi does not support this function.

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SPACE	Invalid *mapSpace* specified.
VI_ERROR_INV_OFFSET	Invalid *offset* specified.
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.
VI_ERROR_TMO	**viMapAddress** could not acquire resource or perform mapping before the timer expired.
VI_ERROR_INV_SIZE	Invalid size of window specified.
VI_ERROR_ALLOC	Unable to allocate window of at least the requested size.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
VI_ERROR_INV_SETUP	Unable to start operation because the setup is invalid (due to attributes being set to an inconsistent state).

See Also **viUnmapAddress**

viMapTrigger

Syntax

```
viMapTrigger(ViSession vi, ViInt16 trigSrc,
  ViInt16 trigDest, ViUInt16 mode);
```

Description

Map the specified trigger source line to the specified destination line. This operation can be used to map one trigger line to another. This operation is valid only on VXI Backplane (BACKPLANE) sessions.

If this operation is called multiple times on the same BACKPLANE resource with the same source trigger line and different destination trigger lines, the result should be that when the source trigger line is asserted all specified destination trigger lines should also be asserted.

If this operation is called multiple times on the same BACKPLANE resource with different source trigger lines and the same destination trigger line the result should be that when any of the specified source trigger lines is asserted, the destination trigger line should also be asserted.

However, mapping a trigger line (as either source or destination) multiple times requires special hardware capabilities and is not guaranteed to be implemented.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
trigSrc	IN	`ViInt16`	Source line from which to map.
trigDest	IN	`ViInt16`	Destination line to which to map.
mode	IN	`ViUInt16`	Specifies the trigger mapping mode. This should always be `VI_NULL` for VISA 2.2.

Special Values for *trgSrc* and *trigDest* Parameter

Value	Action Description
`VI_TRIG_TTL0 - VI_TRIG_TTL7`	Map the specified VXI TTL trigger line.
`VI_TRIG_ECL0 - VI_TRIG_ECL1`	Map the specified VXI ECL trigger line.
`VI_TRIG_PANEL_IN`	Map the controller's front panel trigger input line.
`VI_TRIG_PANEL_OUT`	Map the controller's front panel trigger output line.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.
`VI_SUCCESS_TRIG_MAPPED`	The path from *trigSrc* to *trigDest* is already mapped.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given vi does not support this function.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before operation completed.
`VI_ERROR_INV_MODE`	The value specified by the *mode* parameter is invalid.
`VI_ERROR_LINE_IN_USE`	One of the specified lines (*trigSrc* or *trigDest*) is currently in use.
`VI_ERROR_INV_LINE`	One of the specified lines (*trigSrc* or *trigDest*) is invalid.

Error Codes	Description
VI_ERROR_NSUP_LINE	One of the specified lines (*trigSrc* or *trigDest*) is not supported by this VISA implementation.

See Also

BACKPLANE Resource Description

viMemAlloc

Syntax

```
viMemAlloc(ViSession vi, ViBusSize size,
   ViBusAddress offset);
```

NOTE

This function is *not* implemented in Agilent VISA.

Description

This function returns an offset into a device's memory region that has been allocated for use by this session. If the device to which the given *vi* refers is located on the local interface card, the memory can be allocated either on the device itself or on the computer's system memory. The *offset* returned from the **viMemAlloc** operation is the offset address relative to the device's allocated address base for whichever address space into which the given device exports memory.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
size	IN	**ViBusSize**	Specifies the size of the allocation.
offset	OUT	**ViBusAddress**	Returns the offset of the allocated device memory.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SIZE	Invalid size specified.
VI_ERROR_ALLOC	Unable to allocate shared memory block of the requested size.
VI_ERROR_MEM_NSHARED	The device does not export any memory.

See Also **viMemFree**

viMemFree

Syntax

```
viMemFree(ViSession vi, ViBusAddress offset);
```

NOTE

This function is *not* implemented in Agilent VISA.

Description

This function frees the memory previously allocated using **viMemAlloc**.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
offset	IN	**ViBusAddress**	Specifies the memory previously allocated with **viMemAlloc**.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	The operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_WINDOW_NMAPPED	The specified offset is currently in use by **viMapAddress**.

See Also

viMemAlloc

viMove

Syntax

```
viMove (ViSession vi, ViUInt16 srcSpace,
   ViBusAddress srcOffset, ViUInt16 srcWidth,
   ViUInt16 destSpace, ViBusAddress destOffset,
   ViUInt16 destWidth, ViBusSize length)
```

Description

This operation moves data from the specified source to the specified destination. The source and the destination can either be local memory or the offset of the interface with which this INSTR or MEMACC resource is associated. This operation uses the specified data width and address space.

If the **viSession** parameter (*vi*) refers to an INSTR session, the offset parameters specify relative offsets from the start of the instrument's address space. If the **viSession** parameter (*vi*) refers to a MEMACC session, the offset parameters are absolute offsets from the start of memory in the specified VXI address space.

Valid entries for specifying address space:

Value	Description
VI_A16_SPACE	Address A16 memory address space of the VXI/MXI bus.
VI_A24_SPACE	Address A24 memory address space of the VXI/MXI bus.
VI_A32_SPACE	Address A32 memory address space of the VXI/MXI bus.
VI_LOCAL_SPACE	Address the process-local memory (using virtual address).

Valid entries for specifying widths:

Value	Description
VI_WIDTH_8	Performs an 8-bit (D08) transfer.
VI_WIDTH_16	Performs a 16-bit (D16) transfer.
VI_WIDTH_32	Performs a 32-bit (D32) transfer.

The high-level operation **viMove** operates successfully independently from the low-level operations (**viMapAddress, viPeek8, viPeek16, viPeek32, viPoke8, viPoke16**, and **viPoke32**). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

The *length* specified in the **viMove** operation is the number of elements (of the size corresponding to the operation) to transfer, beginning at the specified offset. Therefore, *offset* + *length*size* cannot exceed the amount of memory exported by the device in the given space.

If *srcSpace* is not **VI_LOCAL_SPACE**, *srcOffset* is a relative address of the device associated with the given INSTR resource. Similarly, if *destspace* is not **VI_LOCAL_SPACE**, *destOffset* is a relative address of the device associated with the given INSTR resource. *srcOffset* and *destOffset* specified in the **viMove** operation for a MEMACC resource are absolute addresses.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
srcSpace	IN	**ViUInt16**	Specifies the address space of the source.
srcOffset	IN	**ViBusAddress**	Offset of the starting address or register from which to read.
srcWidth	IN	**ViUInt16**	Specifies the data width of the source.
destSpace	IN	**ViUInt16**	Specifies the address space of the destination.
destOffset	IN	**ViBusAddress**	Specifies the address space of the destination
destWidth	IN	**ViUInt16**	Specifies the data width of the destination.
length	IN	**ViBusSize**	Number of data elements to transfer, where the data width of the elements to transfer is identical to the source data width.

Return Values

Type **ViStatus** This is the operational return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus Error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid source or destination address specified.
VI_ERROR_INV_OFFSET	Invalid source or destination offset specified.
VI_ERROR_INV_WIDTH	Invalid source or destination width specified.
VI_ERROR_NSUP_OFFSET	Specified source or destination offset is not accessible from this hardware.
VI_ERROR_NSUP_VAR_WIDTH	Cannot support source and destination widths that are different.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NSUP_WIDTH	Specified width is not supported.
VI_ERROR_NSUP_ALIGH_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_LENGTH	Invalid length specified.

See Also

viMoveAsync. Also, see MEMACC Resource Description.

viMoveAsync

Syntax

```
viMoveAsync (ViSession vi, ViUInt16 srcSpace,
   ViBusAddress srcOffset, ViUInt16 srcWidth,
   ViUInt16 destSpace, ViBusAddress destOffset,
   ViUInt16 destWidth, ViBusSize length,ViJobId jobId)
```

NOTE

This function is implemented synchronously in Agilent VISA.

Description

This operation asynchronously moves data from the specified source to the specified destination. This operation queues up the transfer in the system, then it returns immediately without waiting for the transfer to complete. When the transfer terminates, a **VI_EVENT_IO_COMPLETE** event indicates the status of the transfer.

The operation returns *jobId* which you can use either with **viTerminate** to abort the operation or with **VI_EVENT_IO_COMPLETION** events to identify which asynchronous move operations completed. The source and destination can be either local memory or the offset of the device/interface with which this INSTR or MEMACC Resource is associated. This operation uses the specified data width and address space.

If the **viSession** parameter (*vi*) refers to an INSTR session, the offset parameters specify relative offsets from the start of the instrument's address space. If the **viSession** parameter (*vi*) refers to a MEMACC session, the offset parameters are absolute offsets from the start of memory in the specified VXI address space. Valid entries for specifying address space:

Value	Description
VI_A16_SPACE	Address A16 memory address space of the VXI/MXI bus.
VI_A24_SPACE	Address A24 memory address space of the VXI/MXI bus.
VI_A32_SPACE	Address A32 memory address space of the VXI/MXI bus.
VI_LOCAL_SPACE	Addresses the process-local memory (using virtual address).

Valid entries for specifying widths:

Value	Description
VI_WIDTH_8	Performs an 8-bit (D08) transfer.
VI_WIDTH_16	Performs a 16-bit (D16) transfer.
VI_WIDTH_32	Performs a 32-bit (D32) transfer.

Since an asynchronous I/O request could complete before the **viMoveAsync** operation returns, and the I/O completion event can be distinguished based on the job identifier, an application must be made aware of the job identifier before the first moment that the I/O completion event could possibly occur. Setting the output parameter *jobId* before the data transfer even begins ensures that an application can always match the *jobId* parameter with the **VI_ATTR_JOB_ID** attribute of the I/O completion event.

If you pass **VI_NULL** as the *jobId* parameter to the **viMoveAsync** operation, no *jobId* will be returned. This option may be useful if only one asynchronous operation will be pending at a given time. If multiple jobs are queued at the same time on the same session, an application can use the *jobId* to distinguish the jobs, as they are unique within a session. The value **VI_NULL** is a reserved *jobId* and has a special meaning in **viTerminate**.

The status code **VI_ERROR_RSRC_LOCKED** can be returned either immediately or from the **VI_EVENT_IO_COMPLETION** event.

If *srcSpace* is not **VI_LOCAL_SPACE**, *srcOffset* is a relative address of the device associated with the given INSTR resource. Similarly, if *destspace* is not **VI_LOCAL_SPACE**, *destOffset* is a relative address of the device associated with the given INSTR resource.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
srcSpace	IN	**ViUInt16**	Specifies the address space of the source.
srcOffset	IN	**ViBusAddress**	Offset of the starting address or register from which to read.

Name	Direction	Type	Description
srcWidth	IN	**ViUInt16**	Specifies the data width of the source.
destSpace	IN	**ViUInt16**	Specifies the address space of the destination.
destOffset	IN	**ViBusAddress**	Offset of the starting address or register to write to.
destWidth	IN	**ViUInt16**	Specifies the data width of the destination.
length	IN	**ViBusSize**	Number of data elements to transfer, where the data width of the elements to transfer is identical to the source data width.
jobId	OUT	**ViJobId**	Represents the location of an integer that will be set to the job identifier of this asynchronous move operation. Each time an asynchronous move operation is called, it is assigned a unique job identifier.

Special value for *jobId* parameter:

Value	Description
VI_NULL	Operation does not return a job identifier.

Return Values

Type **ViStatus** This is the operational return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Asynchronous operation completed successfully.
VI_SUCCESS_SYNC	Operation Perfomed synchronously.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_QUEUE	Unable to queue move operation.

See Also

viMove. Also, see the INSTR and MEMACC Resource descriptions.

viMoveIn8, viMoveIn16, and viMoveIn32

Syntax

```
viMoveIn8(ViSession vi, ViUInt16 space, ViBusAddress
   offset, ViBusSize length, ViAUInt8 buf8);

viMoveIn16(ViSession vi, ViUInt16 space,
   ViBusAddress offset, ViBusSize length, ViAUInt16 buf16);

viMoveIn32(ViSession vi, ViUInt16 space, ViBusAddress
   offset, ViBusSize length, ViAUInt32 buf32);
```

Description

This function moves an 8-bit, 16-bit, or 32-bit block of data from the specified memory space (assigned memory base + *offset*) to local memory. This function reads the 8-bit, 16-bit, or 32-bit value from the address space pointed to by *space*. The *offset* must be a valid memory address in the *space*. These functions do not require **viMapAddress** to be called prior to their invocation.

If the **viSession** parameter (*vi*) refers to an INSTR session, the offset parameters specify relative offsets from the start of the instrument's address space. If the **viSession** parameter (*vi*) refers to a MEMACC session, the offset parameters are absolute offsets from the start of memory in the specified VXI address space. The valid entries for specifying address space are:

Value	Description
VI_A16_SPACE	Address the A16 address space of VXI/MXI bus.
VI_A24_SPACE	Address the A24 address space of VXI/MXI bus.
VI_A32_SPACE	Address the A32 address space of VXI/MXI bus.

The **viMoveIn** functions do a block move of memory from a VXI device if **VI_ATTR_SRC_INCREMENT** is **1**. However, they do a FIFO read of a VXI memory location if **VI_ATTR_SRC_INCREMENT** is **0** (zero).

The high-level operations **viIn8**, **viIn16**, and **viIn32** operate successfully independently from the low-level operations (**viMapAddress**, **viPeek8, viPeek16, viPeek32, viPoke8, viPoke16**, and **viPoke32**).

The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

For an INSTR resource, the *offset* is a relative address of the device associated with the given INSTR resource. For a MEMACC resource, the *offset* parameter specifies an absolute address.

The *offset* specified in the **viMoveIn8, viMoveIn16**, and **viMoveIn32** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified.

For example, if *space* specifies **VI_A16_SPACE**, *offset* specifies the offset from the logical address base address of the VXI device specified. If *space* specifies **VI_A24_SPACE** or **VI_A32_SPACE**, *offset* specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

The *length* specified in the **viMoveInXX** operations is the number of elements (of the size corresponding to the operation) to transfer, beginning at the specified *offset*. Therefore, *offset* + *length*size* cannot exceed the amount of memory exported by the device in the given space.

The *length* specified in the **viMoveInXX** operations is the number of elements (of the size corresponding to the operation) to transfer, beginning at the specified *offset*. Therefore, *offset* + *length*size* cannot exceed the total amount of memory available in the given space.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
space	IN	**ViUInt16**	Specifies the address space. (See the following table.)
offset	IN	**ViBusAddress**	Offset (in bytes) of the starting address or register to read from.
length	IN	**ViBusSize**	Number of elements to transfer, where the data width of the elements to transfer is 8-bits for **viMoveIn8**, 16-bits for **viMoveIn16**, or 32-bits for **viMoveIn32**.

Name	Direction	Type	Description
buf8, buf16, or *buf32*	OUT	**ViAUInt8, ViAUInt16, or ViAUInt32**	Data read from bus (8-bits for **viMoveIn8**, 16-bits for **viMoveIn16**, and 32-bits for **viMoveIn32**).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_LENGTH	Invalid length specified.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

See Also

viMoveOut8, viMoveOut16, viMoveOut32, viIn8, viIn16, viIn32

viMoveOut8, viMoveOut16, and viMoveOut32

Syntax

```
viMoveOut8(ViSession vi, ViUInt16 space, ViBusAddress
   offset, ViBusSize length, ViAUInt8 buf8);

viMoveOut16(ViSession vi, ViUInt16 space,ViBusAddress
   offset, ViBusSize length, ViAUInt16 buf16);

viMoveOut32(ViSession vi, ViUInt16 space, ViBusAddress
   offset, ViBusSize length, ViAUInt32 buf32);
```

Description

This function moves an 8-bit, 16-bit, or 32-bit block of data from local memory to the specified memory space (assigned memory base + *offset*). This function writes the 8-bit, 16-bit, or 32-bit value to the address space pointed to by *space*. The *offset* must be a valid memory address in the *space*. This function does not require **viMapAddress** to be called prior to its invocation

If the **viSession** parameter (*vi*) refers to an INSTR session, the offset parameters specify relative offsets from the start of the instrument's address space. If the **viSession** parameter (*vi*) refers to a MEMACC session, the offset parameters are absolute offsets from the start of memory in the specified VXI address space. The valid entries for specifying address space are:

Value	Description
VI_A16_SPACE	Address the A16 address space of VXI/MXI bus.
VI_A24_SPACE	Address the A24 address space of VXI/MXI bus.
VI_A32_SPACE	Address the A32 address space of VXI/MXI bus.

The **viMoveOut** functions do a block move of memory from a VXI device if **VI_ATTR_DEST_INCREMENT** is **1**. However, they do a FIFO read of a VXI memory location if **VI_ATTR_DEST_INCREMENT** is **0** (zero).

The *offset* specified in the **viMoveOut8, viMoveOut16**, and **viMoveOut32** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified.

For example, if *space* specifies **VI_A16_SPACE**, *offset* specifies the offset from the logical address base address of the VXI device specified. If *space* specifies **VI_A24_SPACE** or **VI_A32_SPACE**, *offset* specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

The *length* specified in the **viMoveOutXX** operations is the number of elements (of the size corresponding to the operation) to transfer, beginning at the specified *offset*. Therefore, *offset* + *length*size* cannot exceed the amount of memory exported by the device in the given space.

The *length* specified in the **viMoveOutXX** operations is the number of elements (of the size corresponding to the operation) to transfer, beginning at the specified *offset*. Therefore, *offset* + *length*size* cannot exceed the total amount of memory available in the given space.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
space	IN	**ViUInt16**	Specifies the address space. (See the following table.)
offset	IN	**ViBusAddress**	Offset (in bytes) of the starting address or register to write to.
length	IN	**ViBusSize**	Number of elements to transfer, where the data width of the elements to transfer is 8-bits for **viMoveOut8**, 16-bits for **viMoveOut16**, or 32-bits for **viMoveOut32**.
buf8, buf16, or buf32	IN	**ViAUInt8, ViAUInt16,** or **ViAUInt32**	Data written to bus (8-bits for **viMoveOut8**, 16-bits for **viMoveOut16**, and 32-bits for **viMoveOut32**).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_LENGTH	Invalid length specified.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

See Also

viMoveIn8, viMoveIn16, viMoveIn32, viOut8, viOut16, viOut32

viOpen

Syntax

viOpen(ViSession *sesn*, **ViRsrc** *rsrcName*, **ViAccessMode** *accessMode*, **ViUInt32** *timeout*, **ViSession** *vi*);

Description

This function opens a session to the specified device. It returns a session identifier that can be used to call any other functions to that device.

Parameters

Name	Direction	Type	Description
sesn	IN	**ViSession**	Resource Manager session (should always be the Default Resource Manager for VISA returned from **viOpenDefaultRM**).
rsrcName	IN	**ViRsrc**	Unique symbolic name of a resource. (See the following tables.)
accessMode	IN	**ViAccessMode**	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.)

Name	Direction	Type	Description
timeout	IN	**ViUInt32**	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. (**M**ust use **VI_NULL** in VISA 1.0.)
vi	OUT	**ViSession**	Unique logical identifier reference to a session.

Address String Grammar for *rsrcName* Parameter

Interface	Syntax
VXI	**VXI[***board***]::***VXI logical address***[::INSTR]**
VXI	**VXI[***board***]::MEMACC**
VXI	**VXI[***board***][::***VXI logical address***]::BACKPLANE**
GPIB-VXI	**GPIB-VXI[***board***]::***VXI logical address***[::INSTR]**
GPIB-VXI	**GPIB-VXI[***board***]::MEMACC**
GPIB-VXI	**GPIB-VXI[***board***][::***VXI logical address***]::BACKPLANE**
GPIB	**GPIB[***board***]::***primary address***[::***secondary address***][::INSTR]**
GPIB	**GPIB[***board***]::INTFC**
ASRL	**ASRL[***board***][::INSTR]**
TCPIP	**TCPIP[***board***]::***host address***[::***LAN device name***]::INSTR**
TCPIP	**TCPIP[***board***]::***host address***::***port***::SOCKET**

Examples of Address Strings for *rsrcName* Parameter

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.
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.
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.

Return Values

Type **`ViStatus`** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Session opened successfully.
`VI_SUCCESS_DEV_NPRESENT`	Session Opened Successfully, but the device at the specified address is not responding.
`VI_WARN_CONFIG_NLOADED`	The specified configuration either does not exist or could not be loaded using VISA-specified defaults.

Error Codes	Description
`VI_ERROR_INV_SESSION` **`VI_ERROR_INV_OBJECT`**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
VI_ERROR_NSUP_OPER	The given *sesn* does not support this function. For VISA, this function is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to open a session.
VI_ERROR_RSRC_BUSY	The resource is valid but VISA cannot currently access it.
VI_ERROR_RSRC_LOCKED	Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_TMO	A session to the resource could not be obtained within the specified timeout period.
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.

See Also **viClose**

viOpenDefaultRM

Syntax

```
viOpenDefaultRM(ViSession sesn);
```

Description

This function returns a session to the Default Resource Manager resource. This function must be called before any VISA functions can be invoked. The first call to this function initializes the VISA system, including the Default Resource Manager resource, and also returns a session to that resource. Subsequent calls to this function return unique sessions to the same Default Resource Manager resource.

NOTE

All devices to be used must be connected and operational prior to the first VISA function call (**viOpenDefaultRM**). The system is configured only on the *first* **viOpenDefaultRM** per process.

If **viOpenDefaultRM** is first called without devices connected and then called again when devices are connected, the devices will not be recognized. You must close **ALL** Resource Manager sessions and reopen with all devices connected and operational.

Parameters

Name	Direction	Type	Description
sesn	OUT	**ViSession**	Unique logical identifier to a Default Resource Manager session.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Session to the Default Resource Manager resource created successfully.

Error Codes	Description
`VI_ERROR_SYSTEM_ERROR`	The VISA system failed to initialize.
`VI_ERROR_ALLOC`	Insufficient system resources to create a session to the Default Resource Manager resource.
`VI_ERROR_INV_SETUP`	Some implementation-specific configuration file is corrupt or does not exist.

See Also **viOpen, viFindRsrc, viClose**

viOut8, viOut16, and viOut32

Syntax

```
viOut8(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViUInt8 val8);

viOut16(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViUInt16 val16);

viOut32(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViUInt32 val32);
```

Description

This function writes an 8-bit, 16-bit, or 32-bit word to the specified memory space (assigned memory base + *offset*). This function takes the 8-bit, 16-bit, or 32-bit value and stores its contents to the address space pointed to by *space*. The *offset* must be a valid memory address in the *space*. This function does not require **viMapAddress** to be called prior to its invocation.

If the **viSession** parameter (*vi*) refers to an INSTR session, the *offset* parameter specifies a relative offset from the start of the instrument's address *space*. If the **viSession** parameter (*vi*) refers to a MEMACC session, the *offset* parameter is an absolute offset from the start of memory in that VXI address *space*. The valid entries for specifying address space are:

Value	Description
VI_A16_SPACE	Address the A16 address space of VXI/MXI bus.
VI_A24_SPACE	Address the A24 address space of VXI/MXI bus.
VI_A32_SPACE	Address the A32 address space of VXI/MXI bus.

The high-level operations **viOut8**, **viOur16**, and **viOut32** operate successfully independently from the low-level operations (**viMapAddress**, **viPeek8, viPeek16, viPeek32, viPoke8, viPoke16**, and **viPoke32**). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

For an INSTR resource, the *offset* is a relative address of the device associated with the given INSTR resource. For a MEMACC resource, the *offset* parameter specifies an absolute address.

The *offset* specified in the **viOut8**, **viOut16**, and **viOut32** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified.

For example, if *space* specifies **VI_A16_SPACE**, *offset* specifies the offset from the logical address base address of the VXI device specified. If *space* specifies **VI_A24_SPACE** or **VI_A32_SPACE**, *offset* specifies the offset from the base address of the VXI device's memory space allocated by the VXI Resource Manager within VXI A24 or A32 space.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
space	IN	**ViUInt16**	Specifies the address space. (See the following table.)
offset	IN	**ViBusAddress**	Offset (in bytes) of the address or register to write to.
val8, *val16*, or *val32*	IN	**ViUInt8**, **ViUInt16**, or **ViUInt32**	Data to write to bus (8-bits for **viOut8**, 16-bits for **viOut16**, and 32-bits for **viOut32**).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.

Error Codes	Description
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid *offset* specified.
VI_ERROR_NSUP_OFFSET	Specified *offset* not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

See Also

viIn8, viIn16, viIn32, viPoke8, viPoke16, viPoke32, viMoveOut8, viMoveOut16, viMoveOut32

viParseRsrc

Syntax

```
viParseRsrc(ViSession sesn, ViRsrc rsrcName,
  VIUInt16 intfType, VIUInt16 intfNum);
```

Description

Parse a resource string to get the interface information. This operation parses a resource string to verify its validity. It should succeed for all strings returned by **viFindRsrc** and recognized by **viOpen**. This operation is useful if you want to know what interface a given resource descriptor would use without actually opening a session to it.

The values returned in *intfType* and *intfNum* correspond to the attributes **VI_ATTR_INTF_TYPE** and **VI_ATTR_INTF_NUM**. These values would be the same if a user opened that resource with **viOpen** and queried the attributes with **viGetAttribute**.

If a VISA implementation recognizes aliases in **viOpen**, it also recognizes those same aliases in **viParseRsrc**. Calling **viParseRsrc** with **"VXI::1::INSTR"** will produce the same results as invoking it with **"vxi::1::instr"**.

NOTE

A VISA implementation should not perform any I/O to the specified resource during this operation. The recommended implementation of **viParseRsrc** will return information determined solely from the resource string and any static configuration information (e.g., .INI files or the Registry).

Parameters

Name	Direction	Type	Description
sesn	IN	`ViSession`	Resource Manager session (should always be the Default Resource Manager for VISA returned from `viOpenDefaultRM`).
rsrcName	IN	`ViRsrc`	Unique symbolic name of a resource.
intfType	OUT	`VIUInt16`	Interface type of the given resource string.
intfNum	OUT	`VIUInt16`	Board number of the interface of the given resource string.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Resource string is valid.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *sesn* does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.
`VI_ERROR_INV_RSRC_NAME`	Invalid resource reference specified. Parsing error.
`VI_ERROR_RSRC_NFOUND`	Insufficient location information or resource not present in the system.
`VI_ERROR_ALLOC`	Insufficient system resources to parse the string.
`VI_ERROR_LIBRARY_NFOUND`	A code library required by VISA could not be located or loaded.

Error Codes	Description
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.

See Also **viFindRsrc, viOpen**

viPeek8, viPeek16, and viPeek32

Syntax

```
viPeek8(ViSession vi, ViAddr addr, ViUInt8 val8);

viPeek16(ViSession vi, ViAddr addr, ViUInt16 val16);

viPeek32(ViSession vi, ViAddr addr, ViUInt32 val32);
```

Description

This function reads an 8-bit, 16-bit, or 32-bit value from the address location specified in *addr*. The address must be a valid memory address in the current process mapped by a previous **viMapAddress** call.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
addr	IN	**ViAddr**	Specifies the source address to read the value.
val8, *val16*, or *val32*	OUT	**ViUInt8**, **ViUInt16**, or **ViUInt32**	Data read from bus (8-bits for **viPeek8**, 16-bits for **viPeek16**, and 32-bits for **viPeek32**).

NOTE

ViAddr is defined as a **void ***. To do pointer arithmetic, you must cast this to an appropriate type (**ViUInt8**, **ViUInt16**, or **ViUInt32**). Then, be sure the offset is correct for the type of pointer you are using. For example, (**ViUInt8 ***)**addr + 4** points to the same location as (**ViUInt16 ***)**addr + 2**.

Return Values

None.

See Also

viPoke8, viPoke16, viPoke32, viMapAddress, viIn8, viIn16, viIn32

viPoke8, viPoke16, and viPoke32

Syntax

```
viPoke8(ViSession vi, ViAddr addr, ViUInt8 val8);

viPoke16(ViSession vi, ViAddr addr, ViUInt16 val16);

viPoke32(ViSession vi, ViAddr addr, ViUInt32 val32);
```

Description

This function takes an 8-bit, 16-bit, or 32-bit value and stores its content to the address pointed to by *addr*. The address must be a valid memory address in the current process mapped by a previous **viMapAddress** call.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
addr	IN	**ViAddr**	Specifies the destination address to store the value.
val8, val16 or *val32*	IN	**ViUInt8, ViUInt16,** or **ViUInt32**	Data written to bus (8-bits for **viPoke8**, 16-bits for **viPoke16**, and 32-bits for **viPoke32**).

NOTE

ViAddr is defined as a **void *** . To do pointer arithmetic, you must cast this to an appropriate type (**ViUInt8, ViUInt16**, or **ViUInt32**). Then, be sure the offset is correct for the type of pointer you are using. For example, (**ViUInt8 ***)**addr + 4** points to the same location as (**ViUInt16 ***)**addr + 2**.

Return Values

None.

See Also

viPeek8, viPeek16, viPeek32, viMapAddress, viOut8, viOut16, viOut32

viPrintf

Syntax

```
viPrintf(ViSession vi, ViString writeFmt, arg1, arg2,...);
```

Description

This function converts, formats, and sends the parameters *arg1, arg2, ...* to the device as specified by the format string. Before sending the data, the function formats the *arg* characters in the parameter list as specified in the *writeFmt* string. You should not use the **viWrite** and **viPrintf** functions in the same session.

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

The *writeFmt* string can include regular character sequences, special formatting characters, and special format specifiers. The regular characters (including white spaces) are written to the device unchanged. The special characters consist of \ (backslash) followed by a character. The format specifier sequence consists of % (percent) followed by an optional modifier (*flag*), followed by a format code.

Up to four *arg* parameters may be required to satisfy a % format conversion request. In the case where multiple *args* are required, they appear in the following order:

- *field width* (* with %d, %f, or %s) if used
- *precision* (* with %d, %f, or %s) if used
- *array_size* (* with %b, %B, %y, %d, or %f) if used
- value to convert

This assumes that a * is provided for both the field width and the precision modifiers in a %s, %d, or %f. The third arg parameter is used to satisfy a ",*" comma operator. The fourth *arg* parameter is the value to be converted itself.

For ANSI C compatibility the following conversion codes are also supported for output codes. These codes are 'i,' 'o,' 'u,' 'n,' 'x,' 'X,' 'e,' 'E,' 'g,' 'G,' and 'p.' For further explanation of these conversion codes, see the *ANSI C Standard*.

Special Formatting Characters

Special formatting character sequences send special characters. The following table lists the special characters and describes what they send to the device.

\n	Sends the ASCII LF character. The END identifier will also be automatically sent.
\r	Sends an ASCII CR character.
\t	Sends an ASCII TAB character.
\###	Sends the ASCII character specified by the octal value.
\"	Sends the ASCII double-quote (") character.
****	Sends a backslash (\) character.

Format Specifiers

The format specifiers convert the next parameter in the sequence according to the modifier and format code, after which the formatted data is written to the specified device. The format specifier has the following syntax:

%[modifiers**]***format code*

where *format code* specifies which data type in which the argument is represented. The *modifiers* are optional codes that describe the target data. In the following tables, a **d** *format code* refers to all conversion codes of type integer (**d, i, o, u, x, X**), unless specified as **%d** only. Similarly, an **f** *format code* refers to all conversion codes of type float (**f, e, E, g, G**), unless specified as **%f** only.

Every conversion command starts with the **%** character and ends with a conversion character (*format code*). Between the **%** character and the *format code*, the *modifiers* in the following tables can appear in the sequence.

ANSI C Standard Modifiers

Modifier	Supported with Format Code	Description
An integer specifying *field width.*	**d, f, s** format codes	This specifies the minimum field width of the converted argument. If an argument is shorter than the field width, it will be padded on the left (or on the right if the **-** flag is present). An asterisk (*****) may be present in lieu of a field width modifier, in which case an extra *arg* is used. This *arg* must be an integer representing the *field width*. Special case: For the **@H, @Q,** and **@B** flags, the *field width* includes the **#H, #!,** and **#B** strings, respectively.
An integer specifying *precision*.	**d, f, s** format codes	The precision string consists of a string of decimal digits. A . (decimal point) must prefix the *precision* string. An asterisk (*****) may be present in lieu of a *precision* modifier, in which case an extra *arg* is used. This *arg* must be an integer representing the precision of a numeric field. The *precision* string specifies the following: ■ The minimum number of digits to appear for the **@1, @H, @Q,** and **@B** flags and the **i, o, u, x,** and **X** format codes. ■ The maximum number of digits after the decimal point in case of **f** format codes. ■ Maximum numbers of characters for the string (**s**) specifier. ■ Maximum significant digits for **g** format code.
An argument length modifier. **h, l, L, z,** and **Z** are legal values. (**z** and **Z** are not ANSI C standard flags.)	**h** (**d, b, B** format codes) **l** (**d, f, b, B** format codes) **L** (**f** format codes) **z, Z** (**b, B** format codes)	The argument length modifiers specify one of the following: ■ The **h** modifier promotes the argument to a short or unsigned short, depending on the format code type. ■ The **l** modifier promotes the argument to a long or unsigned long. ■ The **L** modifier promotes the argument to a long double parameter. ■ The **z** modifier promotes the argument to an array of floats. ■ The **Z** modifier promotes the argument to an array of doubles.

Enhanced Modifiers to ANSI C Standards

Modifier	Supported with Format Code	Description
A comma (,) followed by an integer *n*, where *n* represents the array size.	**%d** (plus variants) and **%f** only	The corresponding argument is interpreted as a reference to the first element of an array of size n. The first n elements of this list are printed in the format specified by the format code. An asterisk (*****) may be present after the , modifier, in which case an extra *arg* is used. This *arg* must be an integer representing the array size of the given type.
@1	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (e.g., **123**).
@2	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (e.g., **123.45**).
@3	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined NR3 compatible number. An NR3 number is a floating point number represented in an exponential form (e.g., **1.2345E-67**).
@H	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <HEXADECIMAL NUMERIC RESPONSE DATA>. The number is represented in a base of sixteen form. Only capital letters should represent numbers. The number is of the form **#H***XXX*.., where *XXX*.. is a hexadecimal number (e.g., **#HAF35B**).
@Q	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <OCTAL NUMERIC RESPONSE DATA>. The number is represented in a base of eight form. The number is of the form **#Q***YYY*.., where *YYY*.. is an octal number (e.g., **#Q71234**).
@B	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <BINARY NUMERIC RESPONSE DATA>. The number is represented in a base two form. The number is of the form **#B***ZZZ*.., where *ZZZ*.. is a binary number (e.g., **#B011101001**).

The following are the allowed format code characters. A format specifier sequence should include one and only one format code.

Standard ANSI C Format Codes

`%` Send the ASCII percent (`%`) character.

`c` Argument type: A character to be sent.

`d` Argument type: An integer.

Modifier	Interpretation
Default functionality	Print integer in NR1 format (integer without a decimal point).
@2 or @3	The integer is converted into a floating point number and output in the correct format.
field width	Minimum field width of the output number. Any of the six IEEE 488.2 modifiers can also be specified with *field width*.
Length modifier `l`	*arg* is a long integer.
Length modifier `h`	*arg* is a short integer.
, array size	*arg* points to an array of integers (or long or short integers, depending on the length modifier) of size *array size*. The elements of this array are separated by *array size* – 1 commas and output in the specified format.

`f` Argument type: A floating point number.

Modifier	Interpretation
Default functionality	Print a floating point number in NR2 format (a number with at least one digit after the decimal point).
@1	Print an integer in NR1 format. The number is truncated.
@3	Print a floating point number in NR3 format (scientific notation). *Precision* can also be specified.
field width	Minimum field width of the output number. Any of the six IEEE 488.2 modifiers can also be specified with *field width*.
Length modifier `l`	*arg* is a double float.

Modifier	Interpretation
Length modifier **`L`**	*arg* is a long double.
, array size	*arg* points to an array of floats (or doubles or long doubles), depending on the length modifier) of size *array size*. The elements of this array are separated by *array size* – 1 commas and output in the specified format.

`s` Argument type: A reference to a NULL-terminated string that is sent to the device without change.

Enhanced Format Codes

`b` Argument type: A location of a block of data.

Flag or Modifier	Interpretation
Default functionality	The data block is sent as an IEEE 488.2 <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>. A count (long integer) must appear as a flag that specifies the number of elements (by default, bytes) in the block. A *field width* or *precision* modifier is not allowed with this format code.
* (asterisk)	An asterisk may be present instead of the count. In such a case, two *args* are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second *arg* is a reference to the data block. The size of an element is determined by the optional length modifier (see below), the default being byte width.
Length modifier **`h`**	The data block is assumed to be an array of unsigned short integers (16-bits). The count corresponds to the number of words rather than bytes. The data is swapped and padded into standard IEEE 488.2 (big endian) format if native computer representation is different.
Length modifier **`l`**	The data block is assumed to be an array of unsigned long integers. The count corresponds to the number of longwords (32-bits). Each longword data is swapped and padded into standard IEEE 488.2 (big endian) format if native computer representation is different.

Flag or Modifier	Interpretation
Length modifier **z**	The data block is assumed to be an array of floats. The count corresponds to the number of floating point numbers (32-bits). The numbers are represented in IEEE 754 (big endian) format if native computer representation is different.
Length modifier **Z**	The data block is assumed to be an array of doubles. The count corresponds to the number of double floats (64-bits). The numbers are represented in IEEE 754 (big endian) format if native computer representation is different.

B Argument type: A location of a block of data. The functionality is similar to **b**, except the data block is sent as an IEEE 488.2 <INDEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>. This format involves sending an ASCII LF character with the END indicator set after the last byte of the block.

y Argument Type: A location of block binary data.

Flag or Modifier	Interpretation
Default functionality	The data block is sent as raw binary data. A count (long integer) must appear as a flag that specifies the number of elements (by default, bytes) in the block. A *field width* or *precision* modifier is not allowed with this format code.
***** (asterisk)	An asterisk may be present instead of the count. In such a case, two *args* are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second *arg* is a reference to the data block. The size of an element is determined by the optional length modifier (see below), the default being byte width.
Length modifier **h**	The data block is an array of unsigned short integers (16-bits). The count corresponds to the number of words rather than bytes. If the optional **!o1** byte order modifier is present, the data is sent in little endian format. Otherwise, the data is sent in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different.

Flag or Modifier	Interpretation
Length Modifier `l`	The data block is an array of unsigned long integers (32 bits) . The count corresponds to the number of longwords rather than bytes. If the optional `!ol` byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different.
Byte order modifier `!ob`	Data is sent in standard IEE 488.2 (big endian) format. This is the default behavior if neither `!ob` nor `!ol` is present.
Byte order modifier `!ol`	Data is sent in little endian format.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
writeFmt	IN	`ViString`	String describing the format for arguments.
arg1, arg2	IN	N/A	Parameters format string is applied to.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Parameters were successfully formatted.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform write function because of I/O error.

Error Codes	Description
VI_ERROR_TMO	Timeout expired before write function completed.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *writeFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viVPrintf**

viQueryf

Syntax

```
viQueryf(ViSession vi, ViString writeFmt,
  ViString readFmt, arg1, arg2,...);
```

Description

This function performs a formatted write and read through a single operation invocation. This function provides a mechanism of "Send, then receive" typical to a command sequence from a commander device. In this manner, the response generated from the command can be read immediately.

This function is a combination of the **viPrintf** and **viScanf** functions. The first *n* arguments corresponding to the first format string are formatted by using the *writeFmt* string and then sent to the device. The write buffer is flushed immediately after the write portion of the operation completes. After these actions, the response data is read from the device into the remaining parameters (starting from parameter *n* + 1) using the *readFmt* string.

This function returns the same VISA status codes as **viPrintf**, **viScanf**, and **viFlush**.

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

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
writeFmt	IN	**ViString**	**ViString** describing the format of the write arguments.
readFmt	IN	**ViString**	**ViString** describing the format of the read arguments.
arg1, arg2	IN OUT	N/A	Parameters on which write and read format strings are applied.

Return Values

Type **`ViStatus`** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Successfully completed the Query operation.

Error Codes	Description
`VI_ERROR_INV_SESSION` **`VI_ERROR_INV_OBJECT`**	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read/write operation because of I/O error.
`VI_ERROR_TMO`	Timeout occurred before read/write operation completed.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* or *readFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	The format specifier is not supported for current argument type.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also

viPrintf, viScanf, viVQueryf

viRead

Syntax

```
viRead(ViSession vi, ViBuf buf, ViUInt32 count,
  ViUInt32 retCount);
```

Description

This function synchronously transfers data from a device. The data that is read is stored in the buffer represented by *buf*. This function returns only when the transfer terminates. Only one synchronous read function can occur at any one time. A **viRead** operation can complete successfully if one or more of the following conditions were met:

- END indicator received
- Termination character read
- Number of bytes read is equal to *count*

It is possible to have one, two, or all three of these conditions satisfied at the same time.

NOTE

You must set specific attributes to make the read terminate under specific conditions. See *Appendix B - VISA Resource Classes*.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	OUT	**ViBuf**	Represents the location of a buffer to receive data from device.
count	IN	**ViUInt32**	Number of bytes to be read.
retCount	OUT	**ViUInt32**	Represents the location of an integer that will be set to the number of bytes actually transferred.

Special value for *retCount* Parameter:

Value	Description
`VI_NULL`	Do not return the number of bytes transferred.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	The function completed successfully and the END indicator was received (for interfaces that have END indicators).
`VI_SUCCESS_TERM_CHAR`	The specified termination character was read.
`VI_SUCCESS_MAX_CNT`	The number of bytes read is equal to *count*.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this function.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before function completed.
`VI_ERROR_RAW_WR_PROT_VIOL`	Violation of raw write protocol occurred during transfer.
`VI_ERROR_RAW_RD_PROT_VIOL`	Violation of raw read protocol occurred during transfer.
`VI_ERROR_OUTP_PROT_VIOL`	Device reported an output protocol error occurred during transfer.
`VI_ERROR_BERR`	Bus error occurred during transfer.
`VI_ERROR_INV_SETUP`	Unable to start read function because setup is invalid (due to attributes being set to an inconsistent state).

Error Codes	Description
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

See Also **viWrite**

viReadAsync

Syntax

```
viReadAsync(ViSession vi, ViBuf buf, ViUInt32 count,
  ViJobId jobId);
```

Description

This function asynchronously transfers data from a device. The data that is read is stored in the buffer represented by *buf*. This function normally returns before the transfer terminates. An I/O Completion event is posted when the transfer is actually completed.

This function returns *jobId*, which you can use either with **viTerminate** to abort the operation or with an I/O Completion event to identify which asynchronous read operation completed.

Since an asynchronous I/O request could complete before the **viReadAsync** operation returns and the I/O completion event can be distinguished based on the job identifier, an application must be made aware of the job identifier before the first moment that the I/O completion event could possibly occur.

Setting the output parameter *jobId* before the data transfer even begins ensures that an application can always match the *jobId* parameter with the **VI_ATTR_JOB_ID** attribute of the I/O completion event.

If you pass **VI_NULL** as the *jobId* parameter to the **viReadAsync** operation, no *jobId* will be returned. This option may be useful if only one asynchronous operation will be pending at a given time. The value **VI_NULL** is a reserved *jobId* and has a special meaning in **viTerminate.**

If multiple jobs are queued at the same time on the same session, an application can use the jobId to distinguish the jobs, as they are unique within a session.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	OUT	**ViBuf**	Represents the location of a buffer to receive data from the device.
count	IN	**ViUInt32**	Number of bytes to be read.

Name	Direction	Type	Description
jobId	OUT	`ViJobId`	Represents the location of a variable that will be set to the job identifier of this asynchronous read operation.

Special value for *jobId* **Parameter:**

Value	Description
`VI_NULL`	Do not return a job identifier.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Asynchronous read operation successfully queued.
`VI_SUCCESS_SYNC`	Read operation performed synchronously.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_QUEUE_ERROR`	Unable to queue read operation.

See Also

viRead, viTerminate, viWrite, viWriteAsync

viReadSTB

Syntax

viReadSTB(ViSession *vi*, **ViUInt16** *status*);

Description

Read a status byte of the service request. This operation reads a service request status from a service requester (the message-based device). For example, on the IEEE 488.2 interface, the message is read by polling devices. For other types of interfaces, a message is sent in response to a service request to retrieve status information.

For a session to a Serial device or TCPIP socket, if **VI_ATTR_IO_PROT** is **VI_PROT_4882_STRS**, the device is sent the string "*STB?\n" and then the device's status byte is read. Otherwise, this operation is not valid. If the status information is only one byte long, the most significant byte is returned with the zero value. If the service requester does not respond in the actual timeout period, **VI_ERROR_TMO** is returned.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to the session.
status	OUT	**ViUInt16**	Service request status byte.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_SRQ_NOCCURRED	Service request has not been received for the session.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

viReadToFile

Syntax

```
viReadToFile (ViSession vi, ViConstString fileName,
  ViUInt32 count, ViUInt32 retCount);
```

Description

Read data synchronously and store the transferred data in a file. This read operation synchronously transfers data. The file specified in *fileName* is opened in binary write-only mode.

If the value of **VI_ATTR_FILE_APPEND_EN** is **VI_FALSE**, any existing contents are destroyed. Otherwise, the file contents are preserved. The data read is written to the file. This operation returns only when the transfer terminates. This operation is useful for storing raw data to be processed later.

VISA uses ANSI C file operations. The mode used by **viReadToFile** is "wb" or "ab" depending on the value of **VI_ATTR_FILE_APPEND_EN**.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
fileName	IN	**ViConstString**	Name of file to which data will be written.
count	IN	**ViUInt32**	Number of bytes to be read.
retCount	OUT	**ViUInt32**	Number of bytes actually transferred.

Special value for *retCount* **Parameter**:

Completion Code	Description
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	The function completed successfully and the END indicator was received (for interfaces that have END indicators).
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to *count*.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read function because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.

Error Codes	Description
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_FILE_ACCESS	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
VI_ERROR_FILE_IO	An error occurred while accessing the specified file.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

See Also **viRead, viWriteFromFile**

viScanf

Syntax

viScanf(ViSession *vi*, **ViString** *readFmt, arg1, arg2,...*);

Description

This operation receives data from a device, formats it by using the format string, and stores the data in the *arg* parameter list. The format string can have format specifier sequences, white space characters, and ordinary characters.

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

The white characters (blank, vertical tabs, horizontal tabs, form feeds, new line/linefeed, and carriage return) are ignored except in the case of **%c** and **%[]**. All other ordinary characters except **%** should match the next character read from the device.

A format specifier sequence consists of a **%**, followed by optional *modifier* flags, followed by one of the *format codes*, in that sequence. It is of the form:

%[modifier**]***format code*

where the optional *modifier* describes the data format, while *format code* indicates the nature of data (data type). One and only one *format code* should be performed at the specifier sequence. A format specification directs the conversion to the next input *arg*.

The results of the conversion are placed in the variable that the corresponding argument points to, unless the asterisk (*****) assignment-suppressing character is given. In such a case, no *arg* is used and the results are ignored.

The **viScanf** function accepts input until an END indicator is read or all the format specifiers in the *readFmt* string are satisfied. It also terminates if the format string character does not match the incoming character. Thus, detecting an END indicator before the *readFmt* string is fully consumed will result in ignoring the rest of the format string.

Also, if some data remains in the buffer after all format specifiers in the *readFmt* string are satisfied, the data will be kept in the buffer and will be used by the next **viScanf** function.

There is a one-to-one correspondence between **%** format conversions and arg parameters in formatted I/O read operations except:

- If a * is present, no *arg* parameters are used.
- If a # is present instead of *field width*, two *arg* parameters are used. The first *arg* is a reference to an integer (**%c, %s, %t, %T**). This *arg* defines the maximum size of the string being read. The second *arg* points to the buffer that will store the read data.
- If a # is present instead of *array_size*, two *arg* parameters are used. The first *arg* is a reference to an integer (**%d, %f**) or a reference to a long integer (**%b, %y**). This *arg* defines the number of elements in the array. The second *arg* points to the array that will store the read data.

If a *size* is present in *field width* for the **%s, %t**, and **%T** format conversions in formatted I/O read operations either as an integer or a # with a corresponding arg, the *size* defines the maximum number of characters to be stored in the resulting string.

For ANSI C compatibility the following conversion codes are also supported for input codes. These codes are 'i,' 'o,' 'u,' 'n,' 'x,' 'X,' 'e,' 'E,' 'g,' 'G,' 'p,' '[...],' and '[^...].' For further explanation of these conversion codes, see the *ANSI C Standard.*

If **`viScanf`** times out, the read buffer is cleared before **`viScanf`** returns. When **`viScanf`** times out, the next call to **`viScanf`** will read from an empty buffer and force a read from the device. The following tables describe optional modifiers that can be used in a format specifier sequence.

ANSI C Standard Modifiers

Modifier	Supported with Format Codes	Description
An integer representing the *field width*	`%s, %c, %[ ]` format codes	It specifies the maximum field width that the argument will take. A `#` may also appear instead of the integer *field width*, in which case the next *arg* is a reference to the *field width*. This *arg* is a reference to an integer for `%c` and `%s`. The *field width* is not allowed for `%d` or `%f`.
A length modifier (`l, h, L, z` or `Z`). `z` and `Z` are not ANSI C standard modifiers.	`h` (`d, b` format codes) `l` (`d, f, b` format codes) `L` (`f` format code) `z, Z` (`b` format code)	The argument length modifiers specify one of the following: ■ The **h** modifier promotes the argument to be a reference to a short integer or unsigned short integer, depending on the format code. ■ The `l` modifier promotes the argument to point to a long integer or unsigned long integer. ■ The `L` modifier promotes the argument to point to a long double floating point parameter. ■ The `z` modifier promotes the argument to point to an array of floats. ■ The `Z` modifier promotes the argument to point to an array of double floats.
`*` (asterisk)	All format codes	An asterisk acts as the assignment suppression character. The input is not assigned to any parameters and is discarded.

Enhanced Modifiers to ANSI C Standards

Modifier	Supported with Format Codes	Description
A comma (`,`) followed by an integer *n*, where *n* represents the array size.	`%d` (plus variants) and `%f` only	The corresponding argument is interpreted as a reference to the first element of an array of size n. The first n elements of this list are printed in the format specified by the conversion character. A number sign (`#`) may be present after the `,` modifier, in which case an extra *arg* is used. This *arg* must be an integer representing the array size of the given type.
`@1`	`%d` (plus variants) and `%f` only	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (e.g., `123`).

Modifier	Supported with Format Codes	Description
@2	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (e.g., **123.45**).
@H	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <HEXADECIMAL NUMERIC RESPONSE DATA>. The number is represented in a base of sixteen form. Only capital letters should represent numbers. The number is of the form **#H***XXX*.., where *XXX*.. is a hexadecimal number (e.g., **#HAF35B**).
@Q	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <OCTAL NUMERIC RESPONSE DATA>. The number is represented in a base of eight form. The number is of the form **#Q***YYY*.., where *YYY*.. is an octal number (e.g., **#Q71234**).
@B	**%d** (plus variants) and **%f** only	Converts to an IEEE 488.2 defined <BINARY NUMERIC RESPONSE DATA>. The number is represented in a base two form. The number is of the form **#B***ZZZ*.., where *ZZZ*.. is a binary number (e.g., **#B011101001**).

ANSI C Format Codes

c Argument type: A reference to a character. White space in the device input stream is *not* ignored when using **c**.

Flags or Modifiers	Interpretation
Default functionality	A character is read from the device and stored in the parameter.
field width	*field width* number of characters are read and stored at the reference location (the default field width is **1**). No NULL character is added at the end of the data block

d Argument type: A reference to an integer.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until an entire number is read. The number read must be in one of the following IEEE 488.2 formats: <DECIMAL NUMERIC PROGRAM DATA", also known as NRf. Flexible numeric representation (NR1, NR2, NR3, ...). <NON-DECIMAL NUMERIC PROGRAM DATA> (**#H, #Q**, and **#B**).
field width	The input number will be stored in a field at least this wide.
Length modifier **l**	*arg* is a reference to a long integer.
Length modifier **h**	*arg* is a reference to a short integer. Rounding is performed according to IEEE 488.2 rules (**0.5** and up).
, array size	*arg* points to an array of integers (or long or short integers, depending on the length modifier) of size *array size*. The elements of this array should be separated by commas. Elements will be read until either *array size* number of elements are consumed or they are no longer separated by commas.

f Argument type: A reference to a floating point number.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until an entire number is read. The number read must be in either IEEE 488.2 formats: <DECIMAL NUMERIC PROGRAM DATA> (NRf), or <NON-DECIMAL NUMERIC PROGRAM DATA> (**#H, #Q,** and **#B**).
field width	The input number will be stored in a field at least this wide.
Length modifier **l**	*arg* is a reference to a double floating point number.
Length modifier **L**	*arg* is a reference to a long double number.

Flags or Modifiers	Interpretation
, array size	*arg* points to an array of floats (or doubles or long doubles, depending on the length modifier) of size *array size*. The elements of this array should be separated by commas. Elements will be read until either *array size* number of elements are consumed or they are no longer separated by commas.

s Argument type: A reference to a string.

Flags or Modifiers	Interpretation
Default functionality	All leading white space characters are ignored. Characters are read from the device into the string until a white space character is read.
field width	This flag gives the maximum string size. If the *field width* contains a **#** sign, two arguments are used. The first argument read gives the maximum string size. The second should be a reference to a string. In the case of *field width* characters already read before encountering a white space, additional characters are read and discarded until a white space character is found. In the case of **#** *field width*, the actual number of characters read are stored back in the integer pointed to by the first argument.

Enhanced Format Codes

b Argument type: A reference to a data array.

Flags or Modifiers	Interpretation
Default functionality	The data must be in IEEE 488.2 <ARBITRARY BLOCK PROGRAM DATA> format. The format specifier sequence should have a flag describing the *array size*, which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the *array size* contains a **#** sign, two arguments are used.

Flags or Modifiers	Interpretation
Default functionality (continued)	The first argument read is a pointer to a long integer specifying the maximum number of elements that the array can hold. The second one should be a reference to an array. Also in this case, the actual number of elements read is stored back in the first argument. In absence of length modifiers, the data is assumed to be of byte-size elements. In some cases, data might be read until an END indicator is read.
Length modifier **h**	The array is assumed to be an array of 16-bit words, and count refers to the number of words. The data read from the interface is assumed to be in IEEE 488.2 (big endian) byte ordering. It will be byte swapped and padded as appropriate to the native computer format.
Length modifier **l**	The array is assumed to be a block of 32-bit longwords rather than bytes, and count refers to the number of longwords. The data read from the interface is assumed to be in IEEE 488.2 (big endian) byte ordering. It will be byte swapped and padded as appropriate to the native computer format.
Length modifier **z**	The data block is assumed to be a reference to an array of floats, and count refers to the number of floating point numbers. The data block received from the device is an array of 32-bit IEEE 754 format floating point numbers.
Length modifier **Z**	The data block is assumed to be a reference to an array of doubles, and the count refers to the number of floating point numbers. The data block received from the device is an array of 64-bit IEEE 754 format floating point numbers.

t Argument type: A reference to a string.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until the first END indicator is received. The character on which the END indicator was received is included in the buffer.
field width	This flag gives the maximum string size. If an END indicator is not received before *field width* number of characters, additional characters are read and discarded until an END indicator arrives. **#***field width* has the same meaning as in **%s**.

`T` Argument type: A reference to a string.

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until the first linefeed character (**`\n`**) is received. The linefeed character is included in the buffer.
field width	This flag gives the maximum string size. If a linefeed character is not received before *field width* number of characters, additional characters are read and discarded until a linefeed character arrives. **`#`***field width* has the same meaning as in **`%s`**.

`Y` Argument Type: A location of block binary data.

Flag or Modifier	Interpretation
Default functionality	The data block is read as raw binary data. The format specifier sequence should have a flag describing the *array size*, which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the *array size* contains a # sign, two arguments are used. The first argument read is a pointer to a long integer specifying the maximum number of elements that the array can hold. The second argument should be a reference to an array. Also, in this case, the actual number of elements read is stored back in the first argument. In the absence of length modifiers, the data is assumed to be of byte-size elements. In some cases, data might be read until an END indicator is read.
Length modifier **`h`**	The data block is assumed to be a reference to an array of unsigned short integers (16-bits). The count corresponds to the number of words rather than bytes. If the optional **`!o1`** byte order modifier is present, the data being read is assumed to be in little endian format; otherwise, the data being read is assumed to be in standard IEE 488.2 format. Data will be byte swapped and padded as appropriate to native computer format.

Flags or Modifiers	Interpretation
Length Modifier `l`	The data block is assumed to be a reference to an array of unsigned long integers (32 bits) . The count corresponds to the number of longwords rather than bytes. If the optional `!ol` byte order modifier is present, the data being read is assumed to be in little endian format. Otherwise, the data being read is assumed to be in standard IEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different.
Byte order modifier `!ob`	Data being read is assumed to be in standard IEE 488.2 (big endian) format. This is the default behavior if neither `!ob` nor `!ol` is present.
Byte order modifier `!ol`	Data being read is assumed to be in little endian format.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
readFmt	IN	`ViString`	String describing the format for arguments.
arg1, arg2	OUT	N/A	A list with the variable number of parameters into which the data is read and the format string is applied.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Data was successfully read and formatted into *arg* parameter(s).

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_IO	Could not perform read function because of I/O error.
VI_ERROR_TMO	Timeout expired before read function completed.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also

viVScanf

viSetAttribute

Syntax

```
viSetAttribute(ViSession/ViEvent/ViFindList vi,
  ViAttr attribute, ViAttrState attrState);
```

Description

This function sets the state of an attribute for the specified session. The **viSetAttribute** operation is used to modify the state of an attribute for the specified session, event, or find list.

If a resource cannot set an optional attribute state and the specified attribute state is valid and the attribute description does not specify otherwise, **viSetAttribute** returns error code **VI_ERROR_NSUP_ATTR_STATE**.

Both **VI_WARN_NSUP_ATTR_STATE** and **VI_ERROR_NSUP_ATTR_STATE** indicate that the specified attribute state is not supported. Unless a specific rule states otherwise, a resource normally returns the error code **VI_ERROR_NSUP_ATTR_STATE** when it cannot set a specified attribute state. The completion code **VI_WARN_NSUP_ATTR_STATE** is intended to alert the application that although the specified optional attribute state is not supported, the application should not fail.

One example is attempting to set an attribute value that would increase performance speeds. This is different than attempting to set an attribute value that specifies required but nonexistent hardware (such as specifying a VXI ECL trigger line when no hardware support exists) or a value that would change assumptions a resource might make about the way data is stored or formatted (such as byte order). See specific attribute descriptions for text that allows the completion code **VI_WARN_NSUP_ATTR_STATE**.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession** **ViEvent** **ViFindList**	Unique logical identifier to a session, event, or find list.
attribute	IN	**ViAttr**	Resource attribute for which the state is modified.

Name	Direction	Type	Description
attrState	IN	`ViAttrState`	The state of the attribute to be set for the specified resource. The interpretation of the individual attribute value is defined by the resource.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Attribute value set successfully.
`VI_WARN_NSUP_ATTR_STATE`	Although the specified attribute state is valid, it is not supported by this resource implementation. (The application will still work, but this may have a performance impact.)

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_NSUP_ATTR`	The specified attribute is not defined by the referenced resource.
`VI_ERROR_NSUP_ATTR_STATE`	The specified state of the attribute is not valid, or is not supported as defined by the resource. (The application probably will not work if this error is returned.)
`VI_ERROR_ATTR_READONLY`	The specified attribute is read-only.

See Also

`viGetAttribute`. See *Appendix B - VISA Resource Classes* for a list of attributes and attribute values.

viSetBuf

Syntax

```
viSetBuf(ViSession vi, ViUInt16 mask, ViUInt32 size);
```

Description

Set the size for the formatted I/O and/or serial communication buffer(s). This operation changes the buffer size of the read and/or write buffer for formatted I/O and/or serial communication. The *mask* parameter specifies whichbuffer to set the size of. The *mask* parameter can specify multiple buffers by bit-ORing any of the following values together.

Flag	Interpretation
VI_READ_BUF	Formatted I/O read buffer.
VI_WRITE_BUF	Formatted I/O write buffer.
VI_IO_IN_BUF	I/O communication receive buffer.
VI_IO_OUT_BUF	I/O communication transmit buffer.

For backward compatibility, **VI_IO_IN_BUF** is the same as **VI_ASRL_IN_BUF** and **VI_IO_OUT_BUF** is the same as **VI_ASRL_OUT_BUF**.

Since not all serial drivers support user-defined buffer sizes, it is possible that a specific implementation of VISA may not be able to control this feature. If an application requires a specific buffer size for performance reasons, but a specific implementation of VISA cannot guarantee that size, it is recommended to use some form of handshaking to prevent overflow conditions.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
mask	IN	**ViUInt16**	Specifies the type of buffer.
size	IN	**ViUInt32**	The size to be set for the specified buffer(s).

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Buffer size set successfully.
VI_WARN_NSUP_BUF	The specified buffer is not supported.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_ALLOC	The system could not allocate the buffer(s) of the specified *size* because of insufficient system resources.
VI_ERROR_INV_MASK	The system cannot set the buffer for the given *mask*.

See Also

viFlush

viSPrintf

Syntax

viSPrintf(**ViSession** *vi*, **ViBuf** *buf*,
ViString *writeFmt, arg1, arg2, ...);*

Description

Same as **viPrintf,** except the data are written to a user-specified buffer rather than the device. This operation is similar to **viPrintf**, except that the output is not written to the device, but is written to the user-specified buffer. This output buffer will be NULL terminated.

If the **viSPrintf** operations outputs an END indicator before all the arguments are satisfied, the rest of the *writeFmt* string will be ignored and the buffer string will still be terminated by a NULL.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	OUT	**ViBuf**	Buffer where data are to be written.
writeFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**.
arg1, arg2	IN	N/A	A list containing the variable number of parameters on which the format string is applied. The formatted data are written to the specified device.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Parameters were successfully formatted.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *writeFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient system resources.

See Also **viPrintf**

viSScanf

Syntax

viSScanf(ViSession *vi*, **ViBuf** *buf*,
ViString *readFmt, arg1, arg2, ...);*

Description

This operation receives data from a user-specified buffer, formats it by using the format string, and stores the data in the *arg* parameter list. The format string can have format specifier sequences, white space characters, and ordinary characters. This function is the same as **viScanf**, except data are read from a user-specified buffer instead of a device.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Buffer from which data are read and formatted.
readFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**.
arg1, arg2	OUT	N/A	A list with the variable number of parameters into which the data are read and the format string is applied.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Data was successfully read and formatted into *arg* parameter(s).

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient system resources.

viStatusDesc

Syntax

```
viStatusDesc(ViSession/ViEvent/ViFindList vi,
  ViStatus status, ViString desc);
```

Description

This function returns a user-readable string that describes the status code passed to the function. If a status code cannot be interpreted by the session, **viStatusDesc** returns the warning **VI_WARN_UNKNOWN_STATUS**.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession** **ViEvent** **ViFindList**	Unique logical identifier to a session, event, or find list.
status	IN	**ViStatus**	Status code to interpret.
desc	OUT	**ViString**	The user-readable string interpretation of the status code passed to the function. Must be at least 256 characters to receive output.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	Description successfully returned.
VI_WARN_UNKNOWN_STATUS	The status code passed to the function could not be interpreted.

viTerminate

Syntax

```
viTerminate(ViSession vi, ViUInt16 degree,
  ViJobId jobId);
```

NOTE

This function is *not* implemented in Agilent VISA since all I/O is done synchronously.

Description

This function requests a VISA session to terminate normal execution of an operation. This operation requests a session to terminate normal execution of an operation, as specified by the *jobId* parameter. The *jobId* parameter is a unique value generated from each call to an asynchronous operation.

If a user passes **VI_NULL** as the *jobId* value to **viTerminate**, a VISA implementation should abort any calls in the current process executing on the specified *vi*. Any call that is terminated this way should return **VI_ERROR_ABORT**. Due to the nature of multi-threaded systems, for example where operations in other threads may complete normally before the operation **viTerminate** has any effect, the specified return value is not guaranteed.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to an object.
degree	IN	**ViUInt16**	**VI_NULL**
jobId	IN	**ViJobId**	Specifies an operation identifier.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Request serviced successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_INV_DEGREE`	Invalid degree specified.
`VI_ERROR_INV_JOB_ID`	Invalid job identifier specified.

See Also

viReadAsync, viWriteAsync, viMoveAsync

viUninstallHandler

Syntax

```
viUninstallHandler(ViSession vi, ViEventType eventType,
  ViHndlr handler, ViAddr userHandle);
```

Description

This function allows applications to uninstall handlers for events on sessions. Applications should also specify the value in the *userHandle* parameter that was passed to **viInstallHandler** while installing the handler.

VISA identifies handlers uniquely using the *handler* reference and the *userHandle*. All the handlers, for which the *handler* reference and the *userHandle* matches, are uninstalled.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
eventType	IN	**ViEventType**	Logical event identifier.
handler	IN	**ViHndlr**	Interpreted as a valid reference to a handler to be uninstalled by an application. (See the following table.)
userHandle	IN	**ViAddr**	A value specified by an application that can be used for identifying handlers uniquely in a session for an event.

The following events are valid:

Event Name	Description
VI_EVENT_SERVICE_REQ	Notification that a device is requesting service.
VI_EVENT_VXI_SIGP	Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_TRIG	Notification that a hardware trigger was received from a device.
VI_EVENT_IO_COMPLETION	Notification that an asynchronous operation has completed

Special Values for *handler* Parameter

Value	Action Description
`VI_ANY_HNDLR`	Uninstall all the handlers with the matching value in the *UserHandle* parameter.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Event handler successfully uninstalled.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_INV_EVENT`	Specified event type is not supported by the resource.
`VI_ERROR_INV_HNDLR_REF`	Either the specified handler reference or the user context value (or both) does not match any installed handler.
`VI_ERROR_HNDLR_NINSTALLED`	A handler is not currently installed for the specified event.

See Also

See the handler prototype `viEventHandler` for its parameter description. See the `viEnableEvent` description for information about enabling different event handling mechanisms. See individual event descriptions for context definitions.

viUnlock

Syntax

```
viUnlock(ViSession vi);
```

Description

This function is used to relinquish a lock previously obtained using the **viLock** function.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
VI_SUCCESS	The lock was successfully relinquished.
VI_SUCCESS_NESTED_EXCLUSIVE	The call succeeded, but this session still has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The call succeeded, but this session still has nested shared locks.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given *vi* does not identify a valid session or object.
VI_ERROR_SESN_NLOCKED	The current session did not have any lock on the resource.

See Also

viLock. For more information on locking, see *Chapter 4 - Programming with VISA*.

viUnmapAddress

Syntax

```
viUnmapAddress(ViSession vi);
```

Description

This function unmaps memory space previously mapped by the **viMapAddress** function.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Operation completed successfully.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.

See Also

viMapAddress

viUnmapTrigger

Syntax

```
viUnmapTrigger(ViSession vi, ViInt16 trigSrc,
  ViInt16 trigDest);
```

Description

This operation can be used to map one trigger line to another. This operation is valid only on VXI Backplane (BACKPLANE) sessions.

This operation unmaps only one trigger mapping per call. If **viMapTrigger** was called multiple times on the same BACKPLANE resource and created multiple mappings for either *trigSrc* or *trigDest*, trigger mappings other than the one specified by *trigSrc* and *trigDest* remain in effect after this call completes.

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
trigSrc	IN	ViInt16	Source line used in previous map.
trigDest	IN	ViInt16	Destination line used in previous map.

Special Value for *trgSrc* Parameter

Value	Action Description
VI_TRIG_TTL0 - VI_TRIG_TTL7	Unmap the specified VXI TTL trigger line.
VI_TRIG_ECL0 - VI_TRIG_ECL1	Unmap the specified VXI ECL trigger line.
VI_TRIG_PANEL_IN	Unmap the controller's front panel trigger input line.
VI_TRIG_PANEL_OUT	Unmap the controller's front panel trigger output line.

Special Values for *trigDest* Parameter

Value	Action Description
`VI_TRIG_TTL0 - VI_TRIG_TTL7`	Unmap the specified VXI TTL trigger line.
`VI_TRIG_ECL0 - VI_TRIG_ECL1`	Unmap the specified VXI ECL trigger line.
`VI_TRIG_PANEL_IN`	Unmap the controller's front panel trigger input line.
`VI_TRIG_PANEL_OUT`	Unmap the controller's front panel trigger output line.
`VI_TRIG_ALL`	Unmap all trigger lines to which *trigSrc* is currently connected.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given vi does not support this function.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_INV_LINE`	One of the specified lines (*trigSrc* or *trigDest*) is invalid.
`VI_ERROR_TRIG_MAPPED`	The path from *trigSrc* to *trigDest* is not currently mapped.
`VI_ERROR_NSUP_LINE`	One of the specified lines (*trigSrc* or *trigDest*) is not supported by this VISA implementation.

See Also

BACKPLANE Resource Description

viVPrintf

Syntax

```
viVPrintf(ViSession vi, ViString writeFmt,
  ViVAList params);
```

Description

This function converts, formats, and sends *params* to the device as specified by the format string. This function is similar to **viPrintf**, except that the **ViVAList** parameters list provides the parameters rather than separate *arg* parameters.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
writeFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**. See **viPrintf** for description.
params	IN	**ViVAList**	A list containing the variable number of parameters on which the format string is applied. The formatted data is written to the specified device.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Parameters were successfully formatted.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_IO	Could not perform write function because of I/O error.
VI_ERROR_TMO	Timeout expired before write function completed.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *writeFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viPrintf**

viVQueryf

Syntax

viVQueryf(ViSession *vi*, **ViString** *writeFmt*, **ViString** *readFmt*,**ViVAList** *params*);

Description

This function performs a formatted write and read through a single operation invocation. This function is similar to **viQueryf**, except that the **ViVAList** parameters list provides the parameters rather than the separate *arg* parameter list in **viQueryf**.

NOTE

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

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
writeFmt	IN	**ViString**	The format string is applied to write parameters in **ViVAList**.
readFmt	IN	**ViString**	The format string is applied to read parameters in **ViVAList**.
params	IN OUT	**ViVAList**	A list containing the variable number of write and read parameters. The write parameters are formatted and written to the specified device. The read parameters store the data read from the device after the format string is applied to the data.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Successfully completed the Query operation.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read/write operation because of I/O error.
`VI_ERROR_TMO`	Timeout occurred before read/write operation completed.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* or *readFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	The format specifier is not supported for current argument type.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viVPrintf, viVScanf, viQueryf**

viVScanf

Syntax

viVScanf(ViSession *vi*, **ViString** *readFmt*, **ViVAList** *params*);

Description

This function reads, converts, and formats data using the format specifier and then stores the formatted data in *params*. This function is similar to **viScanf**, except that the **ViVAList** parameters list provides the parameters rather than separate *arg* parameters.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
readFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**. See **viScanf** for description.
params	OUT	**ViVAList**	A list with the variable number of parameters into which the data is read and the format string is applied.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Data was successfully read and formatted into *arg* parameter(s).

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.

Error Codes	Description
VI_ERROR_IO	Could not perform read function because of I/O error.
VI_ERROR_TMO	Timeout expired before read function completed.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viScanf**

viVSPrintf

Syntax

```
viVSPrintf(ViSession vi, ViBuf buf, ViString writeFmt,
   ViVAList params);
```

Description

Same as **viVPrintf**, except data are written to a user-specified buffer rather than a device. This operation is similar to **viVPrintf**, except the output is not written to the device but is written to the user-specified buffer. This output buffer will be NULL terminated.

If the **viVSPrintf** operation outputs an END indicator before all the arguments are satisfied, the rest of the *writeFmt* string will be ignored and the buffer string will still be terminated by a NULL.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	OUT	**ViBuf**	Buffer where data are to be written.
writeFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**.
params	IN	**ViVAList**	A list containing the variable number of parameters on which the format string is applied. The formatted data are written to the specified device.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Parameters were successfully formatted.

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *writeFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viSPrintf, viVPrintf**

viVSScanf

Syntax

```
viVSScanf(ViSession vi, ViBuf buf, ViString readFmt,
   ViVAList params);
```

Description

This function reads, converts, and formats data using the format specifier and then stores the formatted data in *params*. This operation is similar to **viVScanf**, except data are read from a user-specified buffer rather than a device.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Buffer from which data are read and formatted.
readFmt	IN	**ViString**	The format string to apply to parameters in **ViVAList**.
params	OUT	**ViVAList**	A list with the variable number of parameters into which data are read and the format string is applied.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Data were successfully read and formatted into arg parameter(s).

Error Codes	Description
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).

Error Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

See Also **viSScanf, viVScanf**

viVxiCommandQuery

Syntax

```
viVxiCommandQuery(ViSession vi, ViUInt16 mode,
  ViUInt32 cmd, ViUInt32 response);
```

Description

Send the device a miscellaneous command or query and/or retrieve the response to a previous query. This operation can send a command or query or receive a response to a query previously sent to the device. The *mode* parameter specifies whether to issue a command and/or retrieve a response, and what type or size of command and/or response to use.

If the *mode* parameter specifies sending a 16-bit command, the upper half of the *cmd* parameter is ignored. If the *mode* parameter specifies just retrieving a response, the *cmd* parameter is ignored. If the *mode* parameter specifies sending a command only, the *response* parameter is ignored and may be **VI_NULL**. If a response is retrieved but is only a 16-bit value, the upper half of the *response* parameter will be set to 0.

Refer to the *VXI Specification* for defined word serial commands. The command values **Byte Available, Byte Request, Clear,** and **Trigger** are not valid for this operation.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
mode	IN	**ViBuf**	Specifies whether to issue a command and/or retrieve a response. See the Description section for actual values.
cmd	IN	**ViUInt32**	The miscellaneous command to send.
response	OUT	**ViUInt32**	The response retrieved from the device. If the mode specifies sending a command, this parameter may be **VI_NULL**.

Special Values for *mode* Parameter:

Mode	Action Description
`VI_VXI_CMD16`	Send 16-bit Word Serial command.
`VI_VXI_CMD16_RESP16`	Send 16-bit Word Serial query, get 16-bit response.
`VI_VXI_RESP16*`	Get 16-bit response from previous query.
`VI_VXI_CMD32*`	Send 32-bit Word Serial command.
`VI_VXI_CMD32_RESP16*`	Send 32-bit Word Serial query, get 16-bit response.
`VI_VXI_CMD32_RESP32*`	Send 32-bit Word Serial query, get 32-bit response.
`VI_VXI_RESP32*`	Get 32-bit response from previous query.

* Not supported in Agilent VISA

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this function.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before function completed.
`VI_ERROR_RAW_WR_PROT_VIOL`	Violation of raw write protocol occurred during transfer.
`VI_ERROR_RAW_RD_PROT_VIOL`	Violation of raw read protocol occurred during transfer.

Error Codes	Description
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_RESP_PENDING	A previous response is still pending, causing a multiple query error.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.

See Also INSTR Resource Description

viWaitOnEvent

Syntax

```
viWaitOnEvent(ViSession vi, ViEventType inEventType,
  ViUInt32 timeout, ViEventType outEventType,
  ViEvent outContext);
```

Description

This function waits for an occurrence of the specified event for a given session. In particular, this function suspends execution of an application thread and waits for an event *inEventType* for at least the time period specified by *timeout*. See the individual event descriptions for context definitions.

If the specified *inEventType* is **VI_ALL_ENABLED_EVENTS**, the function waits for any event that is enabled for the given session. If the specified *timeout* value is **VI_TMO_INFINITE**, the function is suspended indefinitely to wait for an occurrence of an event.

If the value **VI_TMO_IMMEDIATE** is specified in the *timeout* parameter of **viWaitOnEvent**, application execution is not suspended. This operation can be used to dequeue events from an event queue by setting the *timeout* value to **VI_TMO_IMMEDIATE**.

viWaitOnEvent removes the specified event from the event queue if one that matches the type is available. The process of dequeuing makes an additional space available in the queue for events of the same type.

You must call **viEnableEvent** to enable the reception of events of the specified type before calling **viWaitOnEvent**. **viWaitOnEvent** does not perform any enabling or disabling of event reception.

If the value **VI_NULL** is specified in the *outContext* parameter of **viWaitOnEvent** and the return value is successful, **viClose** is automatically invoked on the event context rather than returning it to the application.

The *outEventType* and *outContext* parameters to the **viWaitOnEvent** operation are optional. They can be used if the event type is known from the *inEventType* parameter or if the *eventContext* is not needed to retrieve additional information.

NOTE

Since system resources are used when waiting for events (**viWaitOnEvent**), the **viClose** function must be called to free up event contexts (*outContext*).

This table lists events and associated read-only attributes implemented by Agilent VISA that can be read to get event information on a specific event. Use the **viReadSTB** function to read the status byte of the service request.

Instrument Control (INSTR) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_SERVICE_REQUEST	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_REQ
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_STOP
	VI_ATTR_SIGP_STATUS_ID	ViUInt16	0 to FFFFh
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIG
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

Memory Access (MEMACC) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_IO_ COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_ COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

GPIB Bus Interface (INTFC) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_GPIB_CIC	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_CIC
	VI_ATTR_GPIB_RECV_CIC_ STATE	ViBoolean	VI_TRUE VI_FALSE
VI_EVENT_GPIB_TALK	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_TALK
VI_EVENT_GPIB_ LISTEN	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_ LISTEN
VI_EVENT_CLEAR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_CLEAR
VI_EVENT_TRIGGER	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIGGER
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_SW
VI_EVENT_IO_ COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_ COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

VXI Mainframe Backplane (BACKPLANE) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_TRIG
	VI_ATTR_RECV_TRIG_ID	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_EVENT_VXI_VME_SYSFAIL	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSFAIL
VI_EVENT_VXI_VME_SYSRESET	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSRESET

TCPIP Socket (SOCKET) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_COMPLETION
	VI_ATTR_STATUS	ViStatus	N/A
	VI_ATTR_JOB_ID	ViJobId	N/A
	VI_ATTR_BUFFER	ViBuf	N/A
	VI_ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh
	VI_ATTR_OPER_NAME	ViString	N/A

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
inEventType	IN	ViEventType	Logical identifier of the event(s) to wait for.
timeout	IN	ViUInt32	Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds.

Name	Direction	Type	Description
outEventType	OUT	`ViEventType`	Logical identifier of the event actually received.
outContext	OUT	`ViEvent`	A handle specifying the unique occurrence of an event.

Special value for *outEventType* **Parameter**:

Value	Description
`VI_NULL`	Do not return the type of event.

Special value for *outContext* **Parameter:**

Value	Description
`VI_NULL`	Do not return an event context.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Wait terminated successfully on receipt of an event occurrence. The queue is empty.
`VI_SUCCESS_QUEUE_NEMPTY`	Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence of the specified *inEventType* type available for this session.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_INV_EVENT`	Specified event type is not supported by the resource.

Error Codes	Description
`VI_ERROR_TMO`	Specified event did not occur within the specified time period.

See Also

See *Chapter 4 - Programming with VISA* for more information on event handling.

viWrite

Syntax

```
viWrite(ViSession vi, ViBuf buf, ViUInt32 count,
  ViUInt32 retCount);
```

Description

This function synchronously transfers data to a device. The data to be written is in the buffer represented by *buf*. This function returns only when the transfer terminates. Only one synchronous write function can occur at any one time.

If you pass **VI_NULL** as the *retCount* parameter to the **viWrite** operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Represents the location of a data block to be sent to device.
count	IN	**ViUInt32**	Specifies number of bytes to be written.
retCount	OUT	**ViUInt32**	Represents the location of an integer that will be set to the number of bytes actually transferred.

Special value for *retCount* Parameter:

Value	Dsscription
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Transfer completed.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start write function because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	Unknown I/O error occurred during transfer.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

See Also

viRead

viWriteAsync

Syntax

```
viWriteAsync(ViSession vi, ViBuf buf, ViUInt32 count,
  ViJobId jobId);
```

> **NOTE**
>
> This function is implemented synchronously in Agilent VISA.

Description

Write data to device asynchronously. This function asynchronously transfers data to a device. The data to be written is in the buffer represented by *buf*. This function normally returns before the transfer terminates. An I/O Completion event is posted when the transfer is actually completed.

This function returns *jobId*, which you can use either with **viTerminate** to abort the operation, or with an I/O Completion event to identify which asynchronous write operation completed.

Since an asynchronous I/O request could complete before the **viWriteAsync** operation returns and the I/O completion event can be distinguished based on the job identifier, an application must be made aware of the job identifier before the first moment that the I/O completion event could possibly occur. Setting the output parameter *jobId* before the data transfer even begins ensures that an application can always match the *jobId* parameter with the **VI_ATTR_JOB_ID** attribute of the I/O completion event.

If you pass **VI_NULL** as the *jobId* parameter to the **viWriteAsync** operation, no *jobId* will be returned. The value **VI_NULL** is a reserved *jobId* and has a special meaning in **viTerminate**. This option may be useful if only one asynchronous operation will be pending at a given time. If multiple jobs are queued at the same time on the same session, an application can use the *jobId* to distinguish the jobs, as they are unique within a session.

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
buf	IN	**ViBuf**	Represents the location of a data block to be sent to the device.

Name	Direction	Type	Description
count	IN	`ViUInt32`	Specifies number of bytes to be written.
jobId	OUT	`ViJobId`	Represents the location of a variable that will be set to the job identifier of this asynchronous write operation.

Special value for *jobId* Parameter:

Value	Description
`VI_NULL`	Do not return a job identifier.

Return Values

Type `ViStatus` This is the function return status. It returns either a completion code or an error code as follows.

Completion Codes	Description
`VI_SUCCESS`	Asynchronous write operation successfully queued.
`VI_SUCCESS_SYNC`	Write operation performed synchronously.

Error Codes	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_QUEUE_ERROR`	Unable to queue write operation.

See Also

viRead, viTerminate, viWrite, viReadAsync

viWriteFromFile

Syntax

```
viWriteFromFile (ViSession vi, ViConstString fileName,
  ViUInt32 count, ViUInt32 retCount);
```

Description

Take data from a file and write it out synchronously. This write operation synchronously transfers data. The file specified in *fileName* is opened in binary read-only mode and the data (up to end-of-file or the number of bytes specified in count) are read. The data is then written to the device. This operation returns only when the transfer terminates.

This operation is useful for sending data that was already processed and/or formatted. VISA uses ANSI C file operations, so the mode used by **viWriteFromFile** is "rb". If you pass **VI_NULL** as the *retCount* parameter to the **viWriteFromFile** operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
fileName	IN	ViConstString	Name of file to which data will be read.
count	IN	ViUInt32	Number of bytes to be written.
retCount	OUT	ViUInt32	Number of bytes actually transferred.

Special value for *retCount* **Parameter**:

Value	Dsscription
VI_NULL	Do not return the number of bytes transferred.

Return Values

Type **ViStatus** This is the function return status. It returns either a completion code or an error code as follows.

Completion Code	Description
VI_SUCCESS	Transfer completed.

Error Codes	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_RW_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_FILE_ACCESS	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
VI_ERROR_FILE_IO	An error occurred while accessing the specified file.
VI_ERROR_CONN_LOST	I/O connection for a session has been lost.

See Also

viWrite, viReadToFile

Notes:

A

VISA Library Information

VISA Library Information

This appendix provides general library information for VISA, including:

- VISA Type Definitions
- VISA Error Codes (Alphabetical)
- VISA Error Codes (by Function)
- VISA Directories Information
- Editing VISA Configuration Information

VISA Type Definitions

VISA Data Type	Type Definition	Description
`ViUInt32`	`unsigned long`	A 32-bit unsigned integer.
`ViPUInt32`	`ViUInt32 *`	The location of a 32-bit unsigned integer.
`ViAUInt32`	`ViUInt32 *`	The location of a 32-bit unsigned integer.
`ViInt32`	`signed long`	A 32-bit signed integer.
`ViPInt32`	`ViInt32 *`	The location of a 32-bit signed integer.
`ViAInt32`	`ViInt32 *`	The location of 32-bit signed integer.
`ViUInt16`	`unsigned short`	A 16-bit unsigned integer.
`ViPUInt16`	`ViUInt16 *`	The location of a 16-bit unsigned integer.
`ViAUInt16`	`ViUInt16 *`	The location of a 16-bit unsigned integer.
`ViInt16`	`signed short`	A 16-bit signed integer.
`ViPInt16`	`ViInt16 *`	The location of a 16-bit signed integer.
`ViAInt16`	`ViInt16 *`	The location of 16-bit signed integer.
`ViUInt8`	`unsigned char`	An 8-bit unsigned integer.
`ViPUInt8`	`ViUInt8 *`	The location of an 8-bit unsigned integer.
`ViAUInt8`	`ViUInt8 *`	The location of an 8-bit unsigned integer.
`ViInt8`	`signed char`	An 8-bit signed integer.
`ViPInt8`	`ViInt8 *`	The location of an 8-bit signed integer.
`ViAInt8`	`ViInt8 *`	The location of an 8-bit signed integer.
`ViAddr`	`void *`	A type that references another data type.
`ViPAddr`	`ViAddr *`	The location of a `ViAddr` .
`ViChar`	`char`	An 8-bit integer representing an ASCII character.
`ViPChar`	`ViChar *`	The location of a `ViChar`.
`ViByte`	`unsigned char`	An 8-bit unsigned integer representing an extended ASCII character.
`ViPByte`	`ViByte *`	The location of a `ViByte`.
`ViBoolean`	`ViUInt16`	A type that is either `VI_TRUE` or `VI_FALSE`.
`ViPBoolean`	`ViBoolean *`	The location of a `ViBoolean`.

VISA Data Type	Type Definition	Description
ViBuf	**ViPByte**	The location of a block of data.
ViPBuf	**ViPByte**	The location of a block of data.
ViString	**ViPChar**	The location of a NULL-terminated ASCII string.
ViPString	**ViPChar**	The location of a NULL-terminated ASCII string.
ViStatus	**ViInt32**	Values that correspond to VISA-defined completion and error codes.
ViPStatus	**ViStatus ***	The location of the completion and error codes.
ViRsrc	**ViString**	A **ViString** type.
ViPRsrc	**ViString**	A **ViString** type.
ViAccessMode	**ViUInt32**	Specifies the different mechanisms that control access to a resource.
ViBusAddress	**ViUInt32**	Represents the system dependent physical address.
ViBusSize	**ViUInt32**	Represents the system dependent physical address size.
ViAttr	**ViUInt32**	Identifies an attribute.
ViVersion	**ViUInt32**	Specifies the current version of the resource.
ViPVersion	**ViVersion ***	The location of **ViVersion**.
ViAttrState	**ViUInt32**	Specifies the type of attribute.
ViPAttrState	**void ***	The location of **ViAttrState**.
ViVAList	**va_list**	The location of a list of variable number of parameters of differing types.
ViEventType	**ViUInt32**	Specifies the type of event.
ViPEventType	**ViEventType ***	The location of a **ViEventType**.
ViEventFilter	**ViUInt32**	Specifies filtering masks or other information unique to an event.
ViObject	**ViUInt32**	Contains attributes and can be closed when no longer needed.
ViPObject	**ViObject ***	The location of a **ViObject**.
ViSession	**ViObject**	Specifies the information necessary to manage a communication channel with a resource.
ViPSession	**ViSession ***	The location of a **ViSession**.

VISA Data Type	Type Definition	Description
`ViFindList`	`ViObject`	Contains a reference to all resources found during a search operation.
`ViPFindList`	`ViFindList *`	The location of a **`ViFindList`**.
`ViEvent`	`ViObject`	Contains information necessary to process an event.
`ViPEvent`	`ViEvent *`	The location of a **`ViEvent`**.
`ViHndlr`	`ViStatus( *) (ViSession# ViEventType# ViEvent# ViAddr)`	A value representing an entry point to an operation for use as a callback.
`ViReal32`	`float`	A 32-bit# single-precision value.
`ViPReal32`	`ViReal32 *`	The location of a 32-bit# single-precision value.
`ViReal64`	`double`	A 64-bit# double-precision value.
`ViPReal64`	`ViReal64 *`	The location of a 64-bit# double-precision value.
`ViJobId`	`ViUInt32`	The location of a variable that will be set to the job identifier.
`ViKeyId`	`ViString`	The location of a string.

VISA Error Codes (Alphabetical)

This table lists VISA completion and error codes in alphabetical order.

Codes	Description
Success Codes	
VI_SUCCESS	Operation completed successfully.
VI_SUCCESS_DEV_NPRESENT	Session opened successfully, but the device at the specified address is not responding.
VI_SUCCESS_EVENT_DIS	The specified event is already disabled.
VI_SUCCESS_EVENT_EN	The specified event is already enabled for at least one of the specified mechanisms.
VI_SUCCESS_MAX_CNT	The number of bytes specified were read.
VI_SUCCESS_NESTED_EXCLUSIVE	The specified access mode was successfully acquired and this session has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The specified access mode was successfully acquired and this session has nested shared locks.
VI_SUCCESS_QUEUE_EMPTY	The event queue was empty while trying to discard queued events.
VI_SUCCESS_QUEUE_NEMPTY	The event queue is not empty.
VI_SUCCESS_SYNC	The read or write operation performed synchronously.
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
Warning Codes	
VI_WARN_CONFIG_NLOADED	The specified configuration either does not exist or could not be loaded using VISA-specified defaults.
VI_WARN_NSUP_ATTR_STATE	The attribute state is not supported by this resource.
VI_WARN_NSUP_BUF	The specified buffer is not supported.
VI_WARN_UNKNOWN_STATUS	The status code passed to the function was unable to be interpreted.
Error Codes	
VI_ERROR_ALLOC	Insufficient system resources to open a session or to allocate the buffer(s) or memory block of the specified size.
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.

Codes	Description
Error Codes (continued)	
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_ATTR_READONLY	The attribute specified is read-only.
VI_ERROR_BERR	A bus error occurred during transfer.
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures for this session.
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event. The session cannot be enabled for the **VI_HNDLR** mode of the callback mechanism.
VI_ERROR_INP_PROT_VIOL	Input protocol error occurred during transfer.
VI_ERROR_INV_ACCESS_KEY	The *requestedKey* value passed in is not a valid access key to the specified resource.
VI_ERROR_INV_ACC_MODE	The access mode specified is invalid.
VI_ERROR_INV_CONTEXT	The event context specified is invalid.
VI_ERROR_INV_DEGREE	The specified degree is invalid.
VI_ERROR_INV_EVENT	The event type specified is invalid for the specified resource.
VI_ERROR_INV_EXPR	The expression specified is invalid.
VI_ERROR_INV_FMT	The format specifier is invalid for the current argument.
VI_ERROR_INV_HNDLR_REF	The specified handler reference and/or the user context value does not match the installed handler.
VI_ERROR_INV_JOB_ID	The specified job identifier is invalid.
VI_ERROR_INV_LENGTH	The length specified is invalid.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_MASK	The system cannot set the buffer for the given mask, or the specified mask does not specify a valid flush operation on the read/write resource.
VI_ERROR_INV_MECH	The mechanism specified for the event is invalid.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_INV_OBJECT	The object reference is invalid.

Codes	Description
Error Codes (continued)	
`VI_ERROR_INV_OFFSET`	The offset specified is invalid.
`VI_ERROR_INV_PARAMETER`	The value of some parameter is invalid.
`VI_ERROR_INV_PROT`	The protocol specified is invalid.
`VI_ERROR_INV_RSRC_NAME`	The resources specified are invalid.
`VI_ERROR_INV_SESSION`	The session specified is invalid.
`VI_ERROR_INV_SETUP`	The setup specified is invalid, possibly due to attributes being set to an inconsistent state, or some implementation-specific configuration file is corrupt or does not exist.
`VI_ERROR_INV_SIZE`	The specified size is invalid.
`VI_ERROR_INV_SPACE`	The address space specified is invalid.
`VI_ERROR_IO`	Could not perform read/write function because of an I/O error, or an unknown I/O error occurred during transfer.
`VI_ERROR_LINE_IN_USE`	The specified trigger line is in use.
`VI_ERROR_MEM_NSHARED`	The device does not export any memory.
`VI_ERROR_NCIC`	The session is referring to something other than the controller in charge.
`VI_ERROR_NIMPL_OPER`	The given operation is not implemented.
`VI_ERROR_NLISTENERS`	No listeners are detected. (Both NRFD and NDAC are deasserted.)
`VI_ERROR_NSUP_ATTR`	The attribute specified is not supported by the specified resource.
`VI_ERROR_NSUP_ATTR_STATE`	The state specified for the attribute is not supported.
`VI_ERROR_NSUP_FMT`	The format specifier is not supported for the current argument type.
`VI_ERROR_NSUP_OFFSET`	The offset specified is not accessible.
`VI_ERROR_NSUP_OPER`	The operation specified is not supported in the given session.
`VI_ERROR_NSUP_WIDTH`	The specified width is not supported by this hardware.
`VI_ERROR_NSYS_CNTLR`	The interface associated with this session is not the system controller.
`VI_ERROR_QUEUE_ERROR`	Unable to queue read or write operation.

Codes	Description
Error Codes (continued)	
`VI_ERROR_OUTP_PROT_VIOL`	Output protocol error occurred during transfer.
`VI_ERROR_RAW_RD_PROT_VIOL`	A violation of raw read protocol occurred during a transfer.
`VI_ERROR_RAW_WR_PROT_VIOL`	A violation of raw write protocol occurred during a transfer.
`VI_ERROR_RSRC_BUSY`	The resource is valid, but VISA cannot currently access it.
`VI_ERROR_RSRC_LOCKED`	The specified operation could not be performed because the resource identifed by *vi* has been locked for this kind of access.
`VI_ERROR_RSRC_NFOUND`	The expression specified does not match any device, or resource was not found.
`VI_ERROR_SRQ_NOCCURED`	A service request has not been received for the session.
`VI_ERROR_SYSTEM_ERROR`	Unknown system error.
`VI_ERROR_TMO`	The operation failed to complete within the specified timeout period.
`VI_ERROR_USER_BUF`	A specified user buffer is not valid or cannot be accessed for the required size.
`VI_ERROR_WINDOW_MAPPED`	The specified session already contains a mapped window.
`VI_ERROR_WINDOW_NMAPPED`	The specified session is not currently mapped.

VISA Error Codes (by Function)

VISA functions are listed in alphabetical order with associated completion and error codes for each function.

viAssertIntrSignal(*vi, mode, statusID*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INTR_PENDING	An interrupt is still pending from a previous call.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_NSUP_INTR	The interface cannot generate an interrupt on the requested level or with the requested *statusID* value.
VI_ERROR_NSUP_MODE	The specified *mode* is not supported by this VISA implementation.

viAssertTrigger(*vi, protocol*);

Codes	Description
VI_SUCCESS	Specified trigger was successfully asserted to the device.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_PROT	The protocol specified is invalid.
VI_ERROR_TMO	Timeout expired before function completed.

Codes	Description
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_LINE_IN_USE	The specified trigger line is currently in use.
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viAssertUtilSignal(*vi*, *line*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_LINE	The value specified by the line parameter is invalid.

viBufRead (*vi, buf, count, retCount*);

Codes	Description
VI_SUCCESS	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to *count*.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_IO	An unknown I/O error occurred during transfer.

viBufWrite (*vi, buf, count, retCount*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

viClear(*vi*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viClose(*vi*);

Codes	Description
VI_SUCCESS	Session closed successfully.
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

viDisableEvent(*vi, eventType, mechanism*);

Codes	Description
VI_SUCCESS	Event disabled successfully.
VI_SUCCESS_EVENT_DIS	Specified event is already disabled for at least one of the specified mechanisms.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

viDiscardEvents(*vi, eventType, mechanism*);

Codes	Description
VI_SUCCESS	Event queue flushed successfully.
VI_SUCCESS_QUEUE_EMPTY	Operation completed successfully, but queue empty.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

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

Codes	Description
VI_SUCCESS	Event enabled successfully.
VI_SUCCESS_EVENT_EN	The specified event is already enabled for at least one of the specified mechanisms.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	The specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.
VI_ERROR_INV_CONTEXT	Invalid event context specified.

Codes	Description
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event. The session cannot be enabled for the **VI_HNDLR** mode of the callback mechanism.

viEventHandler(*vi, eventType, context, userHandle*);

Codes	Description
VI_SUCCESS	Event handled successfully.

viFindNext(*findList, instrDesc*);

Codes	Description
VI_SUCCESS	Resource(s) found.
VI_ERROR_INV_SESSION	The given *findList* is not a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *findList* does not support this function.
VI_ERROR_RSRC_NFOUND	There are no more matches.

viFindRsrc(*sesn, expr, findList, retcnt, instrDesc*);

Codes	Description
VI_SUCCESS	Resource(s) found.
VI_ERROR_INV_SESSION	The given *sesn* is not a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *sesn* does not support this function.
VI_ERROR_INV_EXPR	Invalid expression specified for search.
VI_ERROR_RSRC_NFOUND	Specified expression does not match any devices.

viFlush(*vi, mask*);

Codes	Description
VI_SUCCESS	Buffers flushed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.

Codes	Description
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read/write operation because of I/O error.
`VI_ERROR_TMO`	The read/write operation was aborted because timeout expired while operation was in progress.
`VI_ERROR_INV_MASK`	The specified *mask* does not specify a valid flush operation on read/write resource.

`viGetAttribute`(*vi, attribute, attrState*)**;**

Codes	Description
`VI_SUCCESS`	Resource attribute retrieved successfully.
`VI_ERROR_INV_SESSION`	The given *vi* does not identify a valid session.
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_NSUP_ATTR`	The specified attribute is not defined by the referenced resource.

`viGpibCommand(`*vi*, *buf, count, retCount* **`);`**

Codes	Description
`VI_SUCCESS`	Resource attribute retrieved successfully.
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before operation completed.
`VI_ERROR_INV_SETUP`	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).
`VI_ERROR_NCIC`	The interface associated with this session is not currently the controller in charge.

Codes	Description
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

viGpibControlATN(*vi, mode*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_NSUP_MODE	The specified mode is not supported by this VISA implementation.

viGpibControlREN(*vi*, *mode*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).

Codes	Description
`VI_ERROR_NSYS_CNTLR`	The interface associated with this session is not the system controller.
`VI_ERROR_INV_MODE`	The value specified by the *mode* parameter is invalid.

`viGpibPassControl``(`*vi*, *primAddr, secAddr*`);`

Codes	Description
`VI_SUCCESS`	Operation completed successfully.
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_TMO`	Timeout expired before operation completed.
`VI_ERROR_NCIC`	The interface associated with this session is not currently the controller in charge.
`VI_ERROR_NLISTENERS`	No listeners condition is detected (both NRFD and NDAC are deasserted).
`VI_ERROR_IO`	An unknown I/O error occurred during transfer.

`viGpibSendIFC``(`*vi*`);`

Codes	Description
`VI_SUCCESS`	Operation completed successfully.
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *vi* does not support this operation.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_NSYS_CNTLR`	The interface associated with this session is not the system controller.

viIn8(*vi,space,offset,val8*);
viIn16(*vi,space,offset,val16*);
viIn32(*vi,space,offset,val32*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.

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

Codes	Description
VI_SUCCESS	Event handler installed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not defined by the resource.
VI_ERROR_INV_HNDLR_REF	The given handler reference is invalid.
VI_ERROR_HNDLR_NINSTALLED	The handler was not installed. This may be returned if an application attempts to install multiple handlers for the same event on the same session.

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

Codes	Description
VI_SUCCESS	The specified access mode was successfully acquired.
VI_SUCCESS_NESTED_EXCLUSIVE	The specified access mode was successfully acquired and this session has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The specifed access mode was successfully acquired and this session has nested shared locks.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	The specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_ACCESS_KEY	The *requestedKey* value passed is not a valid access key to the specified resource.
VI_ERROR_TMO	The specified type of lock could not be obtained within the specified timeout period.

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

Codes	Description
VI_SUCCESS	Map successful.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.
VI_ERROR_TMO	Could not acquire resource or perform mapping before the timer expired.

Codes	Description
VI_ERROR_INV_SIZE	Invalid size of window specified.
VI_ERROR_ALLOC	Unable to allocate window of at least the requested size.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viMapTrigger(*vi*, *trigSrc*, *trigDest*, *mode*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_SUCCESS_TRIG_MAPPED	The path from *trigSrc* to *trigDest* is already mapped.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given vi does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.
VI_ERROR_LINE_IN_USE	One of the specified lines (*trigSrc* or *trigDest*) is in use.
VI_ERROR_INV_LINE	One of the specified lines (*trigSrc* or *trigDest*) is invalid.
VI_ERROR_NSUP_LINE	One of the specified lines (*trigSrc* or *trigDest*) is not supported by this VISA implementation.

viMemAlloc(*vi, size, offset*);

Codes	Description
VI_SUCCESS	The operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.

Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SIZE	Invalid size specified.
VI_ERROR_ALLOC	Unable to allocate shared memory block of requested size.
VI_ERROR_MEM_NSHARED	The device does not export any memory.

viMemFree (*vi, offset*) ;

Codes	Description
VI_SUCCESS	The operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_WINDOW_MAPPED	The specified offset is currently in use by **viMapAddress**.

viMove (*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, Length*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_ORDER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid source or destination address space specified.
VI_ERROR_INV_OFFSET	Invalid source or destination offset specified.
VI_ERROR_INV_WIDTH	Invalid source or destination width specified.

Codes	Description
VI_ERROR_NSUP_OFFSET	Invalid source or destination offset is not accessible from this hardware.
VI_ERROR_NSUP_VAR_WIDTH	Cannot support source and destination widths that are different.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_LENGTH	Invalid length specified.

viMoveAsync(*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, Length, jobId*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_SUCCESS_SYNC	Operation performed synchronously.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_ORDER	The given *vi* does not support this operation.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI-ERROR_QUEUE	Unable to queue move operation.

viMoveIn8(*vi, space, offset, length, buf8*);
viMoveIn16(*vi, space, offset,l ength, buf16*);
viMoveIn32(*vi, space, offset, length, buf32*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.

Codes	Description
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_LENGTH	Invalid length specified.
VI_ERROR_NSUP_ALIGN_ OFFSET	the specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viMoveOut8(*vi, space, offset, length, buf8*);
viMoveOut16(*vi, space, offset, length, buf16*);
viMoveOut32(*vi, space, offset, length, buf32*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.

Codes	Description
VI_ERROR_INV_LENGTH	Invalid length specified.
VI_ERROR_NSUP_ALIGN_OFFSET	the specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viOpen(*sesn, rsrcName, accessMode, timeout, vi*) **;**

Codes	Description
VI_SUCCESS	Session opened successfully.
VI_SUCCESS_DEV_NPRESENT	Session opened successfully, but the device at the specified address is not responding.
VI_WARN_CONFIG_NLOADED	The specified configuration either does not exist or could not be loaded using VISA-specified defaults.
VI_ERROR_INV_SESSION	The given *sesn* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *sesn* does not support this function. For VISA, this operation is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to open a session.
VI_ERROR_RSRC_BUSY	The resource is valid, but VISA cannot currently access it.
VI_ERROR_RSRC_LOCKED	Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_TMO	A session to the resource could not be obtained within the specified timeout period.

viOpenDefaultRM(*sesn*);

Codes	Description
VI_SUCCESS	Session to the Default Resource Manager resource created successfully.
VI_ERROR_SYSTEM_ERROR	The VISA system failed to initialize.
VI_ERROR_ALLOC	Insufficient system resources to create a session to the Default Resource Manager resource.
VI_ERROR_INV_SETUP	Some implementation-specific configuration file is corrupt or does not exist.

viOut8(*vi, space, offset, val8*);
viOut16(*vi, space, offset, val16*);
viOut32(*vi, space, offset,v al32*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viParseRsrc(*sesn, rsrcName, intfType, intfNum***);**

Codes	Description
VI_SUCCESS	Resource string is valid.
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *sesn* does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to parse the string.
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.
VI_ERROR_INTF_NUM_ **NCONFIG**	The interface type is valid but the specified interface number is not configured.

viPeek8(*vi, addr, val8*);
viPeek16(*vi, addr, val16*);
viPeek32(*vi, addr, val32*);
These functions do not return any completion or error codes.

viPoke8(*vi, addr, val8*);
viPoke16(*vi, addr, val16*);
viPoke32(*vi, addr, val32*);
These functions do not return any completion or error codes.

viPrintf(*vi, writeFmt, arg1, arg2*);

Codes	Description
VI_SUCCESS	Parameters were successfully formatted.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.

Codes	Description
`VI_ERROR_IO`	Could not perform write operation because of I/O error.
`VI_ERROR_TMO`	Timeout expired before write operation completed.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	A format specifier in the *writeFmt* string is not supported.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

`viQueryf`(*vi, writeFmt, readFmt, arg1, arg2*)**;**

Codes	Description
`VI_SUCCESS`	Successfully completed the Query operation.
`VI_ERROR_INV_SESSION`	The given session is invalid.
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read/write operation because of I/O error.
`VI_ERROR_TMO`	Timeout occurred before read/write operation completed.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* or *readFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	The format specifier is not supported for current argument type.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

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

Codes	Description
`VI_SUCCESS`	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
`VI_SUCCESS_TERM_CHAR`	The specified termination character was read.
`VI_SUCCESS_MAX_CNT`	The number of bytes read is equal to *count*.
`VI_ERROR_INV_SESSION`	The given session is invalid.

Codes	Description
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_IO	An unknown I/O error occurred during transfer.

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

Codes	Description
VI_SUCCESS	Asynchronous read operation successfully queued.
VI_SUCCESS_SYNC	Read operation performed synchronously.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.

Codes	Description
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_QUEUE_ERROR	Unable to queue read operation.

viReadSTB(*vi,status*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_SRQ_NOCCURRED	Service request has not been received for the session.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).

viReadToFile (*vi*, *fileName*, *count*, *retCount*);

Codes	Description
VI_SUCCESS	The function completed successfully and the END indicator was received (for interfaces that have END indicators).
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to *count*.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read function because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_FILE_ACCESS	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.

Codes	Description
`VI_ERROR_FILE_IO`	An error occurred while accessing the specified file.
`VI_ERROR_CONN_LOST`	The I/O connection for the given session has been lost.

`viScanf`(*vi, readFmt, arg1, arg2*)**;**

Codes	Description
`VI_SUCCESS`	Data was successfully read and formatted into *arg* parameter(s).
`VI_ERROR_INV_SESSION`	The given session is invalid.
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_IO`	Could not perform read operation because of I/O error.
`VI_ERROR_TMO`	Timeout expired before read operation completed.
`VI_ERROR_INV_FMT`	A format specifier in the *readFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	A format specifier in the *readFmt* string is not supported.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

`viSetAttribute`(*vi, attribute, attrState*)**;**

Codes	Description
`VI_SUCCESS`	All attribute values set successfully.
`VI_WARN_NSUP_ATTR_STATE`	Although the specified state of the attribute is valid, it is not supported by this resource implementation
`VI_ERROR_INV_SESSION`	The given session is invalid.
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_RSRC_LOCKED`	The specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_NSUP_ATTR`	The specified attribute is not defined by the referenced resource.

Codes	Description
`VI_ERROR_NSUP_ATTR_STATE`	The specified state of the attribute is not valid, or is not supported as defined by the resource.
`VI_ERROR_ATTR_READONLY`	The specified attribute is read-only.

`viSetBuf`(*vi, mask, size*)**;**

Codes	Description
`VI_SUCCESS`	Buffer size set successfully.
`VI_WARN_NSUP_BUF`	The specified buffer is not supported.
`VI_ERROR_INV_SESSION`	The given session is invalid.
`VI_ERROR_INV_OBJECT`	The given object reference is invalid.
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_ALLOC`	The system could not allocate the buffer(s) of the specified size because of insufficient system resources.
`VI_ERROR_INV_MASK`	The system cannot set the buffer for the given *mask*.

`viSPrintf`(*vi*, *buf*, *writeFmt, arg1, arg2, ...*);

Codes	Description
`VI_SUCCESS`	Parameters were successfully formatted.
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	A format specifier in the *writeFmt* string is not supported.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient system resources.

viSScanf(*vi*, *buf*, *readFmt, arg1, arg2, ...);*

Codes	Description
VI_SUCCESS	Data were successfully read and formatted into *arg* parameter(s).
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient system resources.

viStatusDesc(*vi, status, desc*) *;*

Codes	Description
VI_SUCCESS	Description successfully returned.
VI_WARN_UNKNOWN_STATUS	The status code passed to the function could not be interpreted.

viTerminate(*vi, degree, jobId*) *;*

Codes	Description
VI_SUCCESS	Request serviced successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_DEGREE	Invalid degree specified.
VI_ERROR_INV_JOB_ID	Invalid job identifier specified.

viUninstallHandler(*vi, eventType, handler, userHandle*);

Codes	Description
VI_SUCCESS	Event handler successfully uninstalled.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_HNDLR_REF	Either the specified handler reference or the user context value (or both) does not match any installed handler.
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event.

viUnlock(*vi*);

Codes	Description
VI_SUCCESS	The lock was successfully relinquished.
VI_SUCCESS_NESTED_EXCLUSIVE	The call succeeded, but this session still has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The call succeeded, but this session still has nested shared locks.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_SESN_NLOCKED	The current session did not have any lock on the resource.

viUnmapAddress(*vi*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.

viUnmapTrigger(*vi*, *trigSrc*, *trigDest*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given vi does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_LINE	One of the specified lines (*trigSrc* or *trigDest*) is invalid.
VI_ERROR_TRIG_MAPPED	The path from *trigSrc* to *trigDest* is not currently mapped.
VI_ERROR_NSUP_LINE	One of the specified lines (*trigSrc* or *trigDest*) is not supported by this VISA implementation.

viVPrintf(*vi, writeFmt, params*);

Codes	Description
VI_SUCCESS	Parameters were successfully formatted.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_IO	Could not perform write operation because of I/O error.
VI_ERROR_TMO	Timeout expired before write operation completed.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *writeFmt* string is not supported.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

viVQueryf(*vi, writeFmt, readFmt, params*);

Codes	Description
VI_SUCCESS	Successfully completed the Query operation.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_IO	Could not perform read/write operation because of I/O error.
VI_ERROR_TMO	Timeout occurred before read/write operation completed.
VI_ERROR_INV_FMT	A format specifier in the *writeFmt* or *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	The format specifier is not supported for current argument type.
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.

viVScanf(*vi, readFmt, params*);

Codes	Description
VI_SUCCESS	Data were successfully read and formatted into *arg* parameter(s).
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_IO	Could not perform read operation because of I/O error.
VI_ERROR_TMO	Timeout expired before read operation completed.
VI_ERROR_INV_FMT	A format specifier in the *readFmt* string is invalid.
VI_ERROR_NSUP_FMT	A format specifier in the *readFmt* string is not supported.

Codes	Description
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

`viVSPrintf`(*vi, buf, writeFmt, params*)**;**

Codes	Description
`VI_SUCCESS`	Parameters were successfully formatted.
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_INV_FMT`	A format specifier in the *writeFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	A format specifier in the *writeFmt* string is not supported.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

`viVSScanf`(*vi, buf, readFmt, params*)**;**

Codes	Description
`VI_SUCCESS`	Data were successfully read and formatted into *arg* parameter(s).
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_RSRC_LOCKED`	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
`VI_ERROR_INV_FMT`	A format specifier in the *readFmt* string is invalid.
`VI_ERROR_NSUP_FMT`	A format specifier in the *readFmt* string is not supported.
`VI_ERROR_ALLOC`	The system could not allocate a formatted I/O buffer because of insufficient resources.

viVxiCommandQuery(*vi*, *mode*, *cmd*, *response*);

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_RESP_PENDING	A previous response is still pending, causing a multiple query error.
VI_ERROR_INV_MODE	The value specified by the *mode* parameter is invalid.

viWaitOnEvent(*vi, ineventType, timeout, outEventType, outcontext*);

Codes	Description
VI_SUCCESS	Wait terminated successfully on receipt of an event occurrence. The queue is empty.
VI_SUCCESS_QUEUE_NEMPTY	Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence available for this session.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_TMO	Specified event did not occur within the specified time period.

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

Codes	Description
VI_SUCCESS	Transfer completed.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

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

Codes	Description
VI_SUCCESS	Asynchronous write operation successfully queued.
VI_SUCCESS_SYNC	Write operation performed synchronously.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.

Codes	Description
VI_ERROR_QUEUE_ERROR	Unable to queue write operation.

viWriteFromFile(*vi*, *fileName*, *count*, *retCount*);

Codes	Description
VI_SUCCESS	Transfer completed.
VI_ERROR_INV_SESSION **VI_ERROR_INV_OBJECT**	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_RW_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an input protocol error during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	The interface associated with the given *vi* is not currently the controller in charge.
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.
VI_ERROR_FILE_ACCESS	An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights.
VI_ERROR_FILE_IO	An error occurred while accessing the specified file.
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.

VISA Directories Information

This section provides information about the location of VISA software files. This information can be used as reference or for removing VISA software from your system, if necessary.

NOTE

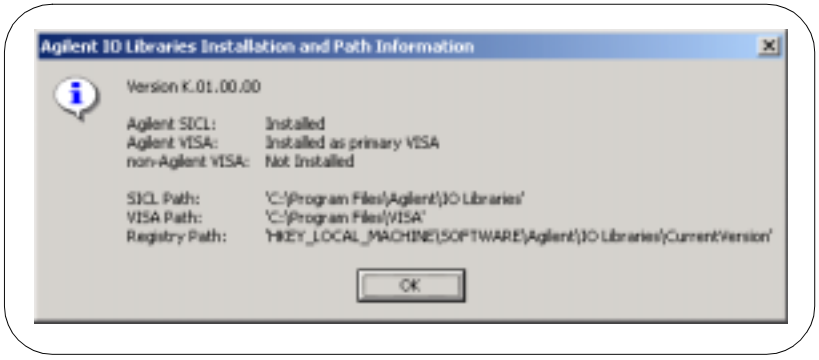
For Windows systems, use “Add/Remove Programs” from the Windows Control Panel to remove files, rather than deleting them manually.

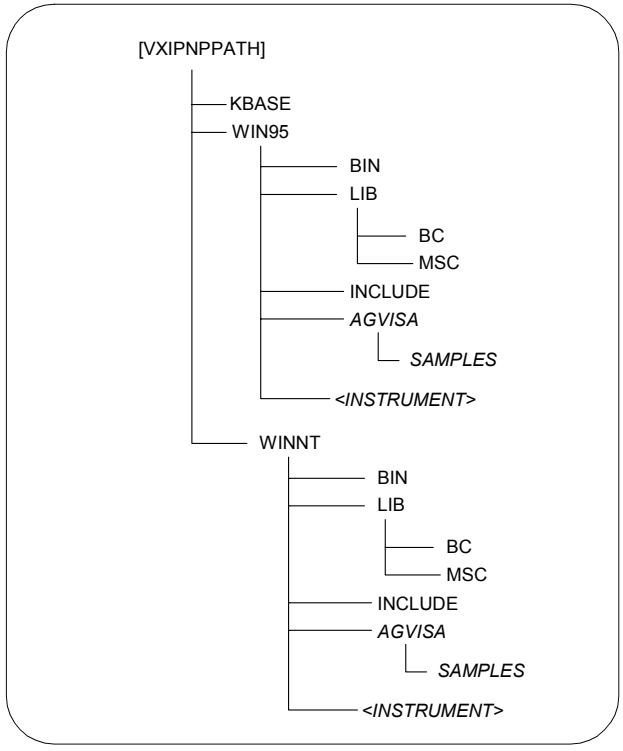
Windows Directory Structure

The VXI*plug&play* alliance defines directory structures to be used with the Windows system framework. As shown in the following figure, VISA files are automatically installed into the WIN95 subdirectory on Windows 95, Windows 98, or Windows Me or into the WINNT subdirectory on Windows 2000 or Windows NT. The [VXIPNPPATH] defaults to \Program Files\VISA, but can be changed during software installation.

The *VISA32.DLL* file is stored in the \WINDOWS\SYSTEM subdirectory (Windows 95, Windows 98, or Windows Me) or in the \WINNT\SYSTEM32 subdirectory (Windows 2000 or Windows NT).

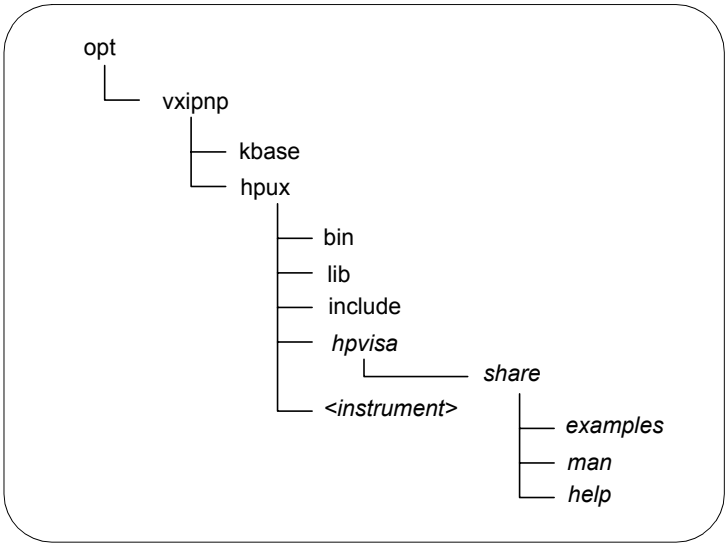
For Windows systems, the VISA path can be displayed by clicking the blue IO icon (near the clock on the Windows taskbar). Then, select **View Documentation** and then **Installation Information** to view a dialog box that contains the VISA path information. A typical display follows.





HP-UX Directory Structure

The VXI*plug&play* alliance defines a directory structure to be used with the UNIX system framework. VISA is automatically installed into the following directory structure on HP-UX 10.20. The [opt] is an optional path that you can change during the software installation.



Editing the VISA Configuration

When the Agilent IO Libraries are configured, certain values are used as defaults in the VISA configuration. In some cases, the default values may affect your system's performance.

If you are having system performance problems, you may need to edit the configuration and change some default values. This section describes how to edit the configuration for VISA on Windows 95, Windows 98, Windows Me, Windows 2000, and Windows NT, and on HP-UX.

Editing on Windows 95/98/Me/2000/NT

When you first configured the Agilent IO Libraries, the default configuration specified that all VISA devices would be identified at runtime. However, this configuration is not ideal for all users.

If you are experiencing performance problems, particularly during **`viFindRsrc`** calls, you may want to change the VISA configuration to identify devices during configuration. This may be especially helpful if you are using a VISA LAN client. To edit the default VISA configuration on Windows 95/98/Me/2000 or Windows NT:

1. If you have not already done so, start Windows 95/98/2000/Me or Windows NT.

2. Run the **IO Config** utility from the **Agilent IO Libraries** program group or from the blue IO icon on the taskbar (use **Run IO Config** from the icon) .

3. Select the interface to be configured from the `Configured Interfaces` box and click the **Edit** button. The **Interface Edit** window is now displayed.

4. Click the **Edit VISA Config** button at the bottom of the window. The dialog box which allows you to add devices is now displayed. You can now manually identify devices by clicking the **Add Device** button and entering the device address.

NOTE

To turn off the default of identifying devices at runtime, unselect the **Identify devices at run-time** box at the top of the dialog box

5. At this time, you may also click the **Auto Add Devices** button at the bottom of the screen to automatically check for devices. If you select this button, the utility will prompt you to make sure all devices are connected and turned on. Once this process is complete, you may edit this list with the **Add Device** and **Remove Device** buttons.

6. Once you have completed adding or removing devices, select the **OK** button to exit the window. Then exit the IO Config utility to save the changes you have made.

Editing on HP-UX

When you first configured the Agilent IO Libraries, the default configuration specified that all VISA devices would be identified at runtime. However, this is not ideal for all users. If you are experiencing performance problems, particularly during **viOpenDefaultRM,** you may want to change the VISA configuration to identify devices during configuration.

To edit the default VISA configuration on HP-UX, use the following command to run the **visacfg** utility:

```
/opt/vxipnp/hpux/hpvisa/visacfg
```

Follow the instructions provided in the utility. When prompted, select the **Add Device** button and add all devices that will be used.

B

VISA Resource Classes

VISA Resource Classes

This appendix describes VISA resource classes, including resource overviews, attributes, events, and operations. This appendix includes:

- Resource Classes Overview
- Instrument Control (INSTR) Resource
- Memory Access (MEMACC) Resource
- GPIB Bus Interface (INTFC) Resource
- VXI Mainframe Backplane (BACKPLANE) Resource
- Servant Device-Side (SERVANT) Resource
- TCPIP Socket (SOCKET) Resource

NOTE

Although the Servant Device-Side (SERVANT) Resource is defined by the *VXIplug&play Systems Alliance Specification* and is described in this Appendix, the SERVANT Resource is not supported in Agilent VISA.

Resource Classes Overview

This section summarizes VISA resource classes and shows applicable interface types for each resource class.

Resource Classes vs. Interface Types

The following table shows the six resource classes that a complete VISA system, fully compliant with the *VXIplug&play Systems Alliance* specification, can implement. Since not all VISA implementations may implement all resource classes for all interfaces, the following table also shows the interfaces applicable to various resource classes.

Resource Class	Interface Types	Resource Class Description
Instrument Control (INSTR)	Generic, GPIB, GPIB-VXI, Serial, TCPIP, VXI	Device operations (reading, writing, triggering, etc.).
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)	Not Supported (GPIB, VXI, TCPIP)	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.

Interface Types vs. Resource Classes

This table shows the five interface types supported by Agilent VISA and the associated Resource Classes for each interface type.

Interface Type	Supported Resource Classes
ASRL	Instrument Control (INSTR)
GPIB	Instrument Control (INSTR) GPIB Bus Interface (INTFC)
GPIB-VXI	Instrument Control (INSTR) Memory Access (MEMACC)
TCPIP	Instrument Control (INSTR) TCPIP Socket (SOCKET)
VXI	Instrument Control (INSTR) Memory Access (MEMACC) VXI Mainframe Backplane (BACKPLANE)

Resource Class Descriptions

The following sections describe each of the six Resource Classes supported by VISA. (As noted, the SERVANT Resource Class is not supported by Agilent VISA. The description for each Resource Class includes:

- Resource Overview
- Resource Attributes
- Resource Events
- Resource Operations (Functions)

NOTE

Attributes are local or global. A local attribute only affects the session specified. A global attribute affects the specified device from any session. Attributes can also be read only (RO) and read/write (RW).

The Generic Attibutes listed apply to all listed interface types. For example, **`VI_ATTR_INTF_NUM`** is listed as a Generic INSTR Resource Attribute, so **`VI_ATTR_INTF_NUM`** applies to the GPIB, GPIB-VXI, VXI, ASRL, and TCPIP interfaces as well.

Instrument Control (INSTR) Resource

This section describes the Instrument Control (INSTR) Resource that is provided to encapsulate the various operations of a device (reading, writing, triggering, etc.).

INSTR Resource Overview

The Instrument Control (INSTR) Resource, like any other resource, defines the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation **`viSetAttribute`**, which is defined in the VISA Resource Template.

Although the INSTR resource does not have **`viSetAttribute`** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task, such as sending a string to a message-based device.

The INSTR Resource lets a controller interact with the device associated with this resource, by providing the controller with services to send blocks of data to the device, request blocks of data from the device, send the device clear command to the device, trigger the device, and find information about the device's status. In addition, it allows the controller to access registers on devices that reside on memory-mapped buses.

INSTR Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic INSTR Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_VXI** **VI_INTF_GPIB** **VI_INTF_GPIB_VXI** **VI_INTF_ASRL** **VI_INTF_TCPIP**	N/A
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_IO_PROT	RW	Local	**ViUInt16**	**VI_NORMAL** **VI_FDC** **VI_HS488** **VI_PROT_488_2_STRS**	**VI_NORMAL**
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	**ViUInt16**	**VI_FLUSH_ON_ACCESS** **VI_FLUSH_DISABLE**	**VI_FLUSH_DISABLE**
VI_ATTR_SEND_END_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_TRUE**
VI_ATTR_SUPPRESS_END_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
VI_ATTR_TERMCHAR	RW	Local	**ViUInt8**	0 to FF_h	$0A_h$ (linefeed)
VI_ATTR_TERMCHAR_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
VI_ATTR_TMO_VALUE	RW	Local	**ViUInt32**	**VI_TMO_IMMEDIATE** 1 to $FFFFFFFE_h$ **VI_TMO_INFINITE**	2000 msec
VI_ATTR_TRIG_ID	RW*	Local	**ViInt16**	**VI_TRIG_SW;** **VI_TRIG_TTL0** to **VI_TRIG_TTL7;** **VI_TRIG_ECL0** to **VI_TRIG_ECL1**	**VI_TRIG_SW**

Attribute Name	Access Privilege		Data Type	Range	Default
Generic INSTR Resource Attributes (continued)					
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL	VI_FLUSH_WHEN_FULL
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_FILE_APPEND_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_GPIB_PRIMARY_ADDR	RO	Global	ViUInt16	0 to 30	N/A
VI_ATTR_GPIB_SECONDARY_ADDR	RO	Global	ViUInt16	0 to 31, VI_NO_SEC_ADDR	N/A
VI_ATTR_GPIB_READDR_EN	RW	Local	Viboolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_GPIB_UNADDR_EN	RW	Local	Viboolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_GPIB_REN_STATE	RO	Global	ViUInt16	VI_STATE_UNKNOWN VI_STATE_ASSERTED VI_STATE_UNASSERTED	N/A
VXI and GPIB-VXI Specific INSTR Resource Attributes					
VI_ATTR_MAINFRAME_LA	RO	Global	ViInt16	0 to 255; VI_UNKNOWN_LA	N/A
VI_ATTR_MANF_ID	RO	Global	ViUInt16	0 to FFF_h	N/A
VI_ATTR_MEM_BASE	RO	Global	ViBusAddress	N/A	N/A
VI_ATTR_MEM_SIZE	RO	Global	ViBusSize	N/A	N/A
VI_ATTR_MEM_SPACE	RO	Global	ViUInt16	VI_A16_SPACE VI_A24_SPACE VI_A32_SPACE	VI_A16_SPACE
VI_ATTR_MODEL_CODE	RO	Global	ViUInt16	0 to $FFFF_h$	N/A
VI_ATTR_SLOT	RO	Global	ViInt16	0 to 12; VI_UNKNOWN_SLOT	N/A
VI_ATTR_VXI_LA	RO	Global	ViInt16	0 to 511	N/A

Attribute Name	Access Privilege		Data Type	Range	Default
VXI and GPIB-VXI Specific INSTR Resource Attributes (continued)					
VI_ATTR_CMDR_LA	RO	Global	ViInt16	0 to 255; VI_UNKNOWN_LA	N/A
VI_ATTR_IMMEDIATE_SERV	RO	Global	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_FDC_CHNL	RW	Local	ViUInt16	0 to 7	N/A
VI_ATTR_FDC_GEN_SIGNAL_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_FDC_MODE	RW	Local	ViUInt16	VI_FDC_NORMAL VI_FDC_STREAM	VI_FDC_NORMAL
VI_ATTR_FDC_USE_PAIR	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_SRC_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_DEST_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_WIN_ACCESS	RO	Local	ViUInt16	VI_NMAPPED VI_USE_OPERS VI_DEREF_ADDR	VI_NMAPPED
VI_ATTR_WIN_BASE_ADDR	RO	Local	ViBusAddress	N/A	N/A
VI_ATTR_WIN_SIZE	RO	Local	ViBusSize	N/A	N/A
VI_ATTR_SRC_BYTE_ORDER	RW	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_ENDIAN
VI_ATTR_DEST_BYTE_ORDER	RW	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_ENDIAN
VI_ATTR_WIN_BYTE_ORDER	RW*	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_ENDIAN

Attribute Name	Access Privilege		Data Type	Range	Default
VXI and GPIB-VXI Specific INSTR Resource Attributes (continued)					
VI_ATTR_SRC_ ACCESS_PRIV	RW	Local	ViUInt16	VI_DATA_NPRIV VI-DATA_PRIV VI_PROG_NPRIV VI_PROG_PRIV VI_BLCK_NPRIV VI_BLCK_PRIV VI_D64_NPRIV VI_D64_PRIV	VI-DATA_ PRIV
VI_ATTR_DEST_ ACCESS_PRIV	RW	Local	ViUInt16	VI_DATA_NPRIV VI-DATA_PRIV VI_PROG_NPRIV VI_PROG_PRIV VI_BLCK_NPRIV VI_BLCK_PRIV VI_D64_NPRIV VI_D64_PRIV	VI-DATA_ PRIV
VI_ATTR_WIN_ ACCESS_PRIV	RW*	Local	ViUInt16	VI_DATA_NPRIV VI-DATA_PRIV VI_PROG_NPRIV VI_PROG_PRIV VI_BLCK_NPRIV VI_BLCK_PRIV	VI-DATA_ PRIV
VI_ATTR_VXI_DEV_ CLASS	RO	Global	ViUInt16	VI_VXI_CLASS_MEMORY VI_VXI_CLASS_EXTENDED VI_VXI_CLASS_MESSAGE VI_VXI_CLASS_REGISTER VI_VXI_CLASS_OTHER	N/A
VI_ATTR_MANF_ NAME	RO	Global	ViString	N/A	N/A
VI_ATTR_MODEL_ NAME	RO	Global	ViString	N/A	N/A
GPIB-VXI Specific INSTR Resource Attribute					
VI_ATTR_INTF_ PARENT_NUM	RO	Global	ViUInt16	0 to $FFFF_h$	VI_ATTR_ INTF_ PARENT_NUM

Attribute Name	Access Privilege		Data Type	Range	Default
ASRL Specific INSTR Resource Attribute					
VI_ATTR_ASRL_AVAIL_NUM	RO	Global	**ViUInt32**	0 to $FFFFFFFF_h$	0
VI_ATTR_ASRL_BAUD	RW	Global	**ViUInt32**	0 to $FFFFFFFF_h$	9600
VI_ATTR_ASRL_DATA_BITS	RW	Global	**ViUInt16**	5 to 8	8
VI_ATTR_ASRL_PARITY	RW	Global	**ViUInt16**	**VI_ASRL_PAR_NONE** **VI_ASRL_PAR_ODD** **VI_ASRL_PAR_EVEN** **VI_ASRL_PAR_MARK** **VI_ASRL_PAR_SPACE**	**VI_ASRL_PAR_NONE**
VI_ATTR_ASRL_STOP_BITS	RW	Global	**ViUInt16**	**VI_ASRL_STOP_ONE** **VI_ASRL_STOP_TWO**	**VI_ASRL_STOP_ONE**
VI_ATTR_ASRL_FLOW_CNTRL	RW	Global	**ViUInt16**	**VI_ASRL_FLOW_NONE** **VI_ASRL_FLOW_XON_XOFF** **VI_ASRL_FLOW_RTS_CTS** **VI_ASRL_FLOW_DTR_DSR**	**VI_ASRL_FLOW_NONE**
VI_ATTR_ASRL_END_IN	RW	Local	**ViUInt16**	**VI_ASRL_END_NONE** **VI_ASRL_END_LAST_BIT** **VI_ASRL_END_TERMCHAR**	**VI_ASRL_END_TERMCHAR**
VI_ATTR_ASRL_END_OUT	RW	Local	**ViUInt16**	**VI_ASRL_END_NONE** **VI_ASRL_END_LAST_BIT** **VI_ASRL_END_TERMCHAR** **VI_ASRL_END_BREAK**	**VI_ASRL_END_NONE**
VI_ATTR_ASRL_CTS_STATE	RO	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_ASRL_DCD_STATE	RO	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_ASRL_DSR_STATE	RO	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_ASRL_RI_STATE	RO	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A

Attribute Name	Access Privilege		Data Type	Range	Default
ASRL Specific INSTR Resource Attribute (continued)					
VI_ATTR_ASRL_DTR_STATE	RW	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_ASRL_RTS_STATE	RW	Global	**ViUInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_ASRL_REPLACE_CHAR	RW	Local	**ViUInt8**	0 to FF_h	0
VI_ATTR_ASRL_XON_CHAR	RW	Local	**ViUInt8**	0 to FF_h	**<Ctrl+Q>** (11_h)
VI_ATTR_ASRL_XOFF_CHAR	RW	Local	**ViUInt8**	0 to FF_h	**<Ctrl+S>** (13_h)
TCPIP Specific INSTR Resource Attributes					
VI_ATTR_TCPIP_ADDR	RW	Global	**ViString**	N/A	N/A
VI_ATTR_TCPIP_HOST_NAME	RW	Global	**ViString**	N/A	N/A
VI_ATTR_TCPIP_DEVICE_NAME	RW	Global	**ViString**	N/A	N/A

* The attribute **VI_ATTR_TRIG_ID** is RW (readable and writeable) when the corresponding session is not enabled to receive trigger events. When the session is enabled to receive trigger events, the attribute **VI_ATTR_TRIG_ID** is RO (read only).

INSTR Resource Attribute Descriptions

Attribute Name	Description
Generic INSTR Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_IO_PROT	Specifies which protocol to use. In VXI systems, for example, you can choose between normal word serial or fast data channel (FDC). In GPIB, you can choose between normal and high-speed (HS488) data transfers. In ASRL systems, you can choose between normal and 488-style transfers, in which case the **viAssertTrigger/viReadSTB/viClear** operations send 488.2-defined strings.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode of the read buffer. When the operational mode is set to **VI_FLUSH_DISABLE** (default), the buffer is flushed only on explicit calls to **viFlush**. If the operational mode is set to **VI_FLUSH_ON_ACCESS**, the buffer is flushed every time a **viScanf** operation completes.
VI_ATTR_SEND_END_EN	Whether to assert END during the transfer of the last byte of the buffer.
VI_ATTR_SUPPRESS_END_EN	Whether to suppress the END indicator termination. If this attribute is set to **VI_TRUE**, the END indicator does not terminate read operations. If this attribute is set to **VI_FALSE**, the END indicator terminates read operations.
VI_ATTR_TERMCHAR	Termination character. When the termination character is read and **VI_ATTR_TERMCHAR_EN** is enabled during a read operation, the read operation terminates.
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a termination character is received.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.
VI_ATTR_TRIG_ID	Identifier for the current triggering mechanism.

Attribute Name	Description
Generic INSTR Resource Attributes (continued)	
`VI_ATTR_WR_BUF_OPER_MODE`	Determines the operational mode of the write buffer. When the operational mode is set to `VI_FLUSH_WHEN_FULL` (default), the buffer is flushed when an END indicator is written to the buffer or when the buffer fills up. If the operational mode is set to `VI_FLUSH_ON_ACCESS`, the write buffer is flushed under the same conditions, and also every time a `viPrintf` operation completes.
`VI_ATTR_DMA_ALLOW_EN`	This attribute specifies whether I/O accesses should use DMA (`VI_TRUE`) or Programmed I/O (`VI_FALSE`). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.
`VI_ATTR_FILE_APPEND_EN`	This attribute specifies whether `viReadToFile` will overwrite (truncate) or append when opening a file.
GPIB and GPIB-VXI Specific INSTR Resource Attributes	
`VI_ATTR_GPIB_PRIMARY_ADDR`	Primary address of the GPIB device used by the given session.
`VI_ATTR_GPIB_SECONDARY_ADDR`	Secondary address of the GPIB device used by the given session.
`VI_ATTR_GPIB_READDR_EN`	This attribute specifies whether to use repeat addressing before each read or write operation.
`VI_ATTR_GPIB_UNADDR_EN`	This attribute specifies whether to unaddress the device (UNT and UNL) after each read or write operation.
`VI_ATTR_GPIB_REN_STATE`	This attribute returns the current state of the GPIB REN interface line.
VXI and GPIB-VXI Specific INSTR Resource Attributes	
`VI_ATTR_MAINFRAME_LA`	This is the logical address of a given device in the mainframe, usually the device with the lowest logical address. Other possible values include the logical address of the Slot 0 controller or of the parent-side extender. Often, these are all the same value. The purpose of this attribute is to provide a unique ID for each mainframe. A VISA manufacturer can choose any of these values, but must be consistent across mainframes. If this value is not known, the attribute value returned is .
`VI_ATTR_MANF_ID`	Manufacturer identification number of the VXIbus device.

Attribute Name	Description
VXI and GPIB-VXI Specific INSTR Resource Attributes (continued)	
VI_ATTR_MEM_BASE	Base address of the device in VXIbus memory address space. This base address is applicable to A24 or A32 address space.
VI_ATTR_MEM_SIZE	Size of memory requested by the device in VXIbus address space.
VI_ATTR_MEM_SPACE	VXIbus address space used by the device. The three types are A16 only, A16/A24, or A16/A32 memory address space.
VI_ATTR_MODEL_CODE	Model code for the device.
VI_ATTR_SLOT	Physical slot location of the VXIbus device. If the slot number is not known, **VI_UNKNOWN_SLOT** is returned.
VI_ATTR_VXI_LA	Logical address of the VXI or VME device used by the given session. For a VME device, the logical address is actually a pseudo-address in the range 256 to 511.
VI_ATTR_CMDR_LA	Logical address of the commander of the VXI device used by the given session.
VI_ATTR_IMMEDIATE_SERV	Specifies whether the given device is an immediate servant of the controller running VISA.
VI_ATTR_FDC_CHNL	This attribute determines which FDC channel will be used to transfer the buffer.
VI_ATTR_FDC_SIGNAL_GEN_EN	Setting this attribute to **VI_TRUE** lets the servant send a signal when control of the FDC channel is passed back to the commander. This action frees the commander from having to poll the FDC header while engaging in an FDC transfer.
VI_ATTR_FDC_MODE	This attribute determines which FDC mode to use (Normal mode or Stream mode).
VI_ATTR_FDC_USE_PAIR	If set to **VI_TRUE**, a channel pair will be used for transferring data. Otherwise, only one channel will be used.
VI_ATTR_SRC_INCREMENT	This is used in the **viMoveInXX** operation to specify how much the source offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the **viMoveInXX** 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.

Attribute Name	Description
VXI and GPIB-VXI Specific INSTR Resource Attributes (continued)	
VI_ATTR_DEST_INCREMENT	This is used in the **viMoveOutXX** operation to specify how much the destination offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the **viMoveOutXX** 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	Modes in which the current window may be accessed: not currently mapped, through operations **viPeekXX** and **viPokeXX** only, or through operations and/or by directly dereferencing 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.
VI_ATTR_SRC_BYTE_ORDER	This attribute specifies the byte order to be used in high-level access operations, such as **viInXX** and **viMoveInXX**, when reading from the source.
VI_ATTR_DEST_BYTE_ORDER	This attribute specifies the byte order to be used in high-level access operations, such as **viOutXX** and **viMoveOutXX**, when writing to the destination
VI_ATTR_WIN_BYTE_ORDER	This attribute specifies the byte order to be used in low-level access operations, such as **viMapAddress**, **viPeekXX** and **viPokeXX**, when accessing the mapped window.
VI_ATTR_SRC_ACCESS_PRIV	This attribute specifies the address modifier to be used in high-level access operations, such as **viInXX** and **viMoveInXX**, when reading from the source.
VI_ATTR_DEST_ACCESS_PRIV	This attribute specifies the address modifier to be used in high-level access operations, such as **viOutXX** and **viMoveOutXX**, when writing to the destination.
VI_ATTR_WIN_ACCESS_PRIV	This attribute specifies the address modifier to be used in low-level access operations, such as **viMapAddress**, **viPeekXX** and **viPokeXX**, when accessing the mapped window.

Attribute Name	Description
VXI and GPIB-VXI Specific INSTR Resource Attributes (continued)	
VI_ATTR_VXI_DEV_CLASS	This attribute represents the VXI-defined device class to which the resource belongs: ■ message based (VI_VXI_CLASS_MESSAGE) ■ register based (VI_VXI_CLASS_REGISTER) ■ extended (VI_VXI_CLASS_EXTENDED) ■ memory (VI_VXI_CLASS_MEMORY) VME devices are usually either register based or belong to a miscellaneous class (VI_VXI_CLASS_OTHER)
VI_ATTR_MANF_NAME	This string attribute is the manufacturer's name. The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can be different between VISA implementations and/or revisions.
VI_ATTR_MODEL_NAME	This string attribute is the model name of the device. The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can be different between VISA implementations and/or revisions.
VI_ATTR_VXI_TRIG_SUPPORT	This attribute shows which VXI trigger lines this implementation supports. This is a bit vector with bits 0-9 corresponding to VI_TRIG_TTL0 through VI_TRIG_ECL1.
GPIB-VXI Specific INSTR Resource Attribute	
VI_ATTR_INTF_PARENT_NUM	Board number of the GPIB board to which the GPIB-VXI is attached.
ASRL Specific INSTR Resource Attributes	
VI_ATTR_ASRL_AVAIL_NUM	This attribute shows the number of bytes available in the global receive buffer.
VI_ATTR_ASRL_BAUD	This is the baud rate of the interface. It is represented as an unsigned 32-bit integer so that any baud rate can be used, but it usually requires a commonly used rate such as 300, 1200, 2400, or 9600 baud.
VI_ATTR_ASRL_DATA_BITS	This is the number of data bits contained in each frame (from 5 to 8). The data bits for each frame are located in the low-order bits of every byte stored in memory.

Attribute Name	Description
ASRL Specific INSTR Resource Attributes (continued)	
VI_ATTR_ASRL_PARITY	This is the parity used with every frame transmitted and received. **VI_ASRL_PAR_MARK** means that the parity bit exists and is always 1. means that the parity bit exists and is always 0.
VI_ATTR_ASRL_STOP_BITS	This is the number of stop bits used to indicate the end of a frame. The value **VI_ASRL_STOP_ONE5** indicates one-and-one-half (1.5) stop bits.
VI_ATTR_ASRL_FLOW_CNTRL	If this attribute is set to **VI_ATTR_ASRL_FLOW_NONE**, the transfer mechanism does not use flow control, and buffers on both sides of the connection are assumed to be large enough to hold all data transferred. If this attribute is set to **VI_ATTR_ASRL_FLOW_XON_XOFF**, the transfer mechanism uses the XON and XOFF characters to perform flow control. The transfer mechanism controls input flow by sending XOFF when the receive buffer is nearly full, and it controls the output flow by suspending transmission when XOFF is received. If this attribute is set to **VI_ATTR_ASRL_FLOW_RTS_CTS**, the transfer mechanism uses the RTS output signal and the CTS input signal to perform flow control. The transfer mechanism controls input flow by unasserting the RTS signal when the receive buffer is nearly full, and it controls output flow by suspending the transmission when the CTS signal is unasserted. If this attribute is set to **VI_ASRL_FLOW_DTR_DSR**, the transfer mechanism uses the DTR output signal and the DSR input signal to perform flow control. The transfer mechanism controls input flow by unasserting the DTR signal when the receive buffer is nearly full, and it controls output flow by suspending the transmission when the DSR signal is unasserted. This attribute can specify multiple flow control mechanisms by bit-ORing multiple values together. However, certain combinations may not be supported by all serial ports and/or operating systems.

Attribute Name	Description
ASRL Specific INSTR Resource Attributes (continued)	
VI_ATTR_ASRL_END_IN	This attribute indicates the method used to terminate read operations. If it is set to **VI_ASRL_END_NONE**, the read will not terminate until all of the requested data is received (or an error occurs). If it is set to **VI_ASRL_END_TERMCHAR**, the read will terminate as soon as the character in **VI_ATTR_TERMCHAR** is received. If it is set to **VI_ASRL_END_LAST_BIT**, the read will terminate as soon as a character arrives with its last bit set. For example, if **VI_ATTR_ASRL_DATA_BITS** is set to 8, then the read will terminate when a character arrives with the 8th bit set.
VI_ATTR_ASRL_END_OUT	This attribute indicates the method used to terminate write operations. If it is set to **VI_ASRL_END_NONE**, the write will not append anything to the data being written. If it is set to **VI_ASRL_END_BREAK**, the write will transmit a break after all the characters for the write have been sent. If it is set to **VI_ASRL_END_LAST_BIT**, the write will send all but the last character with the last bit clear, then transmit the last character with the last bit set. For example, if **VI_ATTR_ASRL_DATA_BITS** is set to 8, then the write will clear the 8th bit for all but the last character, then transmit the last character with the 8th bit set. If it is set to **VI_ASRL_END_TERMCHAR**, the write will send the character in **VI_ATTR_TERMCHAR** after the data being transmitted.
VI_ATTR_ASRL_CTS_STATE	This attribute shows the current state of the Clear To Send (CTS) input signal.
VI_ATTR_ASRL_DCD_STATE	This attribute shows the current state of the Data Carrier Detect (DCD) input signal. The DCD signal is often used by modems to indicate the detection of a carrier (remote modem) on the telephone line. The DCD signal is also known as "Receive Line Signal Detect (RLSD)."
VI_ATTR_ASRL_DSR_STATE	This attribute shows the current state of the Data Set Ready (DSR) input signal.
VI_ATTR_ASRL_DTR_STATE	This attribute is used to manually assert or unassert the Data Terminal Ready (DTR) output signal.
VI_ATTR_ASRL_RI_STATE	This attribute shows the current state of the Ring Indicator (RI) input signal. The RI signal is often used by modems to indicate that the telephone line is ringing.

Attribute Name	Description
ASRL Specific INSTR Resource Attributes (continued)	
`VI_ATTR_ASRL_RTS_STATE`	This attribute is used to manually assert or unassert the Request To Send (RTS) output signal. When the `VI_ATTR_ASRL_FLOW_CNTRL` attribute is set to `VI_ASRL_FLOW_RTS_CTS`, this attribute is ignored when changed, but can be read to determine whether the background flow control is asserting or unasserting the signal.
`VI_ATTR_ASRL_REPLACE_CHAR`	This attribute specifies the character to be used to replace incoming characters that arrive with errors (such as parity error.)
`VI_ATTR_ASRL_XON_CHAR`	This attribute specifies the value of the XON character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.
`VI_ATTR_ASRL_XOFF_CHAR`	This attribute specifies the value of the XOFF character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.
TCPIP Specific INSTR Resource Attributes	
`VI_ATTR_TCPIP_ADDR`	This is the TCPIP address of the device to which the session is connected. This string is formatted in dot-notation.
`VI_ATTR_TCPIP_HOSTNAME`	This specifies the host name of the device. If no host name is available, this attribute returns an empty string.
`VI_ATTR_TCPIP_DEVICE_NAME`	This specifies the LAN device name used by the VXI-11 protocol during connection.

INSTR Resource Events

This resource defines the following events for communication with applications, where AP = Access Privilege.

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 Attributes	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 Attributes	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

VI_EVENT_IO_COMPLETION

Notification that an asynchronous operation has completed.

Event Attributes	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

Event Attributes	Description	AP	Data Type	Range
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_VXI_VME_INTR

Notification that a VXIbus interrupt was received from the device. **NOT IMPLEMENTED IN AGILENT VISA**.

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	**ViEventType**	**VI_EVENT_VXI_VME_INTR**
VI_ATTR_STATUS_ID	32-bit status/ID retrieved during the IACK cycle.	RO	**ViUInt32**	0 to $FFFFFFFF_h$
VI_ATTR_RECV_INTR_LEVEL	VXI interrupt level on which the interrupt was received.	RO	**VIInt16**	1 to 7, **VI_UNKNOWN_LEVEL**

INSTR Resource Operations

viAssertTrigger(*vi, protocol*)
viBufRead(*vi, buf, count, retCount*)
viBufWrite(*vi, buf, count, retCount*)
viClear(*vi*)
viFlush(*vi, mask*)
viGpibControlREN(*vi, mode*)

viIn16(*vi, space, offset, val16*)
viIn32(*vi, space, offset, val32*)
viIn8(*vi, space, offset, val8*)
viMapAddress(*vi, mapSpace, mapBase, mapSize, access, suggested, address*)
viMemAlloc(*vi, size, offset*)
viMemFree(*vi, offset*)

viMove(*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length*)
viMoveAsync(*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId*)
viMoveIn8(*vi, space, offset, length, buf8*)
viMoveIn16(*vi, space, offset, length, buf16*)
viMoveIn32(*vi, space, offset, length, buf32*)

viMoveOut8(*vi, space, offset, length, buf8*)
viMoveOut16(*vi, space, offset, length, buf16*)
viMoveOut32(*vi, space, offset, length, buf32*)
viOut8(*vi, space, offset, val8*)
viOut16(*vi, space, offset, val16*)
viOut32(*vi, space, offset, val32*)
viPeek8(*vi, addr, val8*)
viPeek16(*vi, addr, val16*)
viPeek32(*vi, addr, val32*)
viPoke8(*vi, addr, val8*)
viPoke16(*vi, addr, val16*)
viPoke32(*vi, addr, val32*)

viPrintf(*vi, writeFmt, arg1, arg2, ...*)
viQueryf(*vi, writeFmt, readFmt, arg1, arg2, ...*)
viRead(*vi, buf, count, retCount*)
viReadAsync(*vi, buf, count, jobId*)
viReadSTB(*vi, status*)
viReadToFile(*vi, fileName, count, retCount*)
viScanf(*vi, readFmt, arg1, arg2, ...*)
viSetBuf(*vi, mask, size*)
viSPrintf(*vi, buf, writeFmt, arg1, arg2, ...*)
viSScanf(*vi, buf, readFmt, arg1, arg2, ...*)

viUnmapAddress(*vi*)
viVPrintf(*vi, writeFmt, params*)
viVQueryf(*vi, writeFmt, readFmt, params*)
viVScanf(*vi, readFmt, params*)
viVSPrintf(*vi, buf, writeFmt, params*)
viVSScanf(*vi, buf, readFmt, params*)
viVxiCommandQuery(*vi, mode, cmd, response*)
viWrite(*vi, buf, count, retCount*)
viWriteAsync(*vi, buf, count, jobId*)
viWriteFromFile(*vi, fileName, count, retCount*)

Memory Access (MEMACC) Resource

This section describes the Memory Access (MEMACC) Resource that is provided to encapsulate the address space of a memory-mapped bus, such as the VXIbus.

MEMACC Resource Overview

The Memory Access (MEMACC) Resource encapsulates the address space of a memory-mapped bus such as the VXIbus. A VISA Memory Access Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation **`viSetAttribute`**.

Although the MEMACC resource does not have **`viSetAttribute`** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task, such as reading a register or writing to a memory location.

The MEMACC Resource lets a controller interact with the interface associated with this resource. It does this by providing the controller with services to access arbitrary registers or memory addresses on memory-mapped buses.

MEMACC Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic MEMACC Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_VXI** **VI_INTF_GPIB_VXI**	N/A
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_TMO_VALUE	RW	Local	**ViUInt32**	**VI_TMO_IMMEDIATE** 1 to $FFFFFFFE_h$ **VI_TMO_INFINITE**	2000 msec
VI_ATTR_DMA_ALLOW_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	N/A
VXI and GPIB-VXI Specific MEMACC Resource Attributes					
VI_ATTR_VXI_LA	RO	Global	**ViInt16**	0 to 255	N/A
VI_ATTR_SRC_INCREMENT	RW	Local	**ViInt32**	0 to 1	1
VI_ATTR_DEST_INCREMENT	RW	Local	**ViInt32**	0 to 1	1
VI_ATTR_WIN_ACCESS	RO	Local	**ViUInt16**	**VI_NMAPPED** **VI_USE_OPERS** **VI_DEREF_ADDR**	**VI_NMAPPED**
VI_ATTR_WIN_BASE_ADDR	RO	Local	**ViBusAddress**	N/A	N/A
VI_ATTR_WIN_SIZE	RO	Local	**ViBusSize**	N/A	N/A
VI_ATTR_SRC_BYTE_ORDER	RW	Local	**ViUInt16**	**VI_BIG_ENDIAN** **VI_LITTLE_ENDIAN**	**VI_BIG_ENDIAN**
VI_ATTR_DEST_BYTE_ORDER	RW	Local	**ViUInt16**	**VI_BIG_ENDIAN** **VI_LITTLE_ENDIAN**	**VI_BIG_ENDIAN**
VI_ATTR_WIN_BYTE_ORDER	RW*	Local	**ViUInt16**	**VI_BIG_ENDIAN** **VI_LITTLE_ENDIAN**	**VI_BIG_ENDIAN**

Attribute Name	Access Privilege		Data Type	Range	Default
VXI and GPIB-VXI Specific MEMACC Resource Attributes					
VI_ATTR_SRC_ ACCESS_PRIV	RW	Local	**ViUInt16**	**VI_DATA_NPRIV** **VI_DATA_PRIV** **VI_PROG_NPRIV** **VI_PROG_PRIV** **VI_BLCK_NPRIV** **VI_BLCK_PRIV** **VI_D64_NPRIV** **VI_D64_PRIV**	**VI_DATA_ PRIV**
VI_ATTR_DEST_ ACCESS_PRIV	RW	Local	**ViUInt16**	**VI_DATA_NPRIV** **VI-DATA_PRIV** **VI_PROG_NPRIV** **VI_PROG_PRIV** **VI_BLCK_NPRIV** **VI_BLCK_PRIV** **VI_D64_NPRIV** **VI_D64_PRIV**	**VI_DATA_ PRIV**
VI_ATTR_WIN_ ACCESS_PRIV	RW*	Local	**ViUInt16**	**VI_DATA_NPRIV** **VI-DATA_PRIV** **VI_PROG_NPRIV** **VI_PROG_PRIV** **VI_BLCK_NPRIV** **VI_BLCK_PRIV**	**VI_DATA_ PRIV**
GPIB-VXI Specific MEMACC Resource Attributes					
VI_ATTR_INTF_ PARENT_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	N/A
VI_ATTR_GPIB_ PRIMARY_ADDR	RO	Global	**ViUInt16**	0 to 30	N/A
VI_ATTR_GPIB_ SECONDARY_ADDR	RO	Global	**ViUInt16**	0 to 31, **VI_NO_SEC_ADDR**	N/A

* For VISA 2.2, the attributes **VI_ATTR_WIN_BYTE_ORDER** and **VI_ATTR_WIN_ACCESS_PRIV** are RW (readable and writeable) when the corresponding session is not mapped (**VI_ATTR_WIN_ACCESS = = VI_NMAPPED**). When the session is mapped, these attributes are RO (read only).

MEMACC Resource Attribute Descriptions

Attribute Name	Description
Generic MEMACC Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.
VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/O accesses should use DMA (**VI_TRUE**) or Programmed I/O (**VI_FALSE**). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.
VXI and GPIB-VXI Specific MEMACC Resource Attributes	
VI_ATTR_VXI_LA	Logical address of the local controller.
VI_ATTR_SRC_INCREMENT	This is used in the **viMoveInXX** operation to specify how much the source offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the **viMoveInXX** 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	This is used in the **viMoveOutXX** operation to specify how much the destination offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the **viMoveOutXX** 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.

Attribute Name	Description
VXI and GPIB-VXI Specific MEMACC Resource Attributes (continued)	
`VI_ATTR_WIN_ACCESS`	Modes in which the current window may be accessed: not currently mapped, through operations **`viPeekXX`** and **`viPokeXX`** only, or through operations and/or by directly dereferencing 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.
`VI_ATTR_SRC_BYTE_ORDER`	This attribute specifies the byte order to be used in high-level access operations, such as **`viInXX`** and **`viMoveInXX`**, when reading from the source.
`VI_ATTR_DEST_BYTE_ORDER`	This attribute specifies the byte order to be used in high-level access operations, such as **`viOutXX`** and **`viMoveOutXX`**, when writing to the destination.
`VI_ATTR_WIN_BYTE_ORDER`	This attribute specifies the byte order to be used in low-level access operations, such as **`viMapAddress`**, **`viPeekXX`** and **`viPokeXX`**, when accessing the mapped window.
`VI_ATTR_SRC_ACCESS_PRIV`	This attribute specifies the address modifier to be used in high-level access operations, such as **`viInXX`** and **`viMoveInXX`**, when reading from the source.
`VI_ATTR_DEST_ACCESS_PRIV`	This attribute specifies the address modifier to be used in high-level access operations, such as **`viOutXX`** and **`viMoveOutXX`**, when writing to the destination.
`VI_ATTR_WIN_ACCESS_PRIV`	This attribute specifies the address modifier to be used in low-level access operations, such as **`viMapAddress`**, **`viPeekXX`** and **`viPokeXX`**, when accessing the mapped window.
GPIB-VXI Specific MEMACC Resource Attributes	
`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 given session.
`VI_ATTR_GPIB_SECONDARY_ADDR`	Secondary address of the GPIB-VXI controller used by the given session.

MEMACC Resource Events

This resource defines the following event for communication with applications, where AP = Access Privilege.

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

MEMACC Resource Operations

viIn8(*vi, space, offset, val8*)
viIn16(*vi, space, offset, val16*)
viIn32(*vi, space, offset, val32*)
viMapAddress(*vi, mapSpace, mapBase, mapSize, access, suggested, address*)
viMove(*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length*)
viMoveAsync(*vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId*)

viMoveIn8(*vi, space, offset, length, buf8*)
viMoveIn16(*vi, space, offset, length, buf16*)
viMoveIn32(*vi, space, offset, length, buf32*)
viMoveOut8(*vi, space, offset, length, buf8*)
viMoveOut16(*vi, space, offset, length, buf16*)
viMoveOut32(*vi, space, offset, length, buf32*)

viOut8(*vi, space, offset, val8*)
viOut16(*vi, space, offset, val16*)
viOut32(*vi, space, offset, val32*)
viPeek8(*vi, addr, val8*)
viPeek16(*vi, addr, val16*)
viPeek32(*vi, addr, val32*)
viPoke8(*vi, addr, val8*)
viPoke16(*vi, addr, val16*)
viPoke32(*vi, addr, val32*)
viUnmapAddress(*vi*)

GPIB Bus Interface (INTFC) Resource

This section describes the GPIB Bus Interface (INTFC) Resource that is provided to encapsulate the operations and properties of a raw GPIB interface (reading, writing, triggering, etc.).

INTFC Resource Overview

A VISA GPIB Bus Interface (INTFC) Resource, like any other resource, defines the basic operations and attributes of the VISA Resource Template.

For example, modifying the state of an attribute is done via the operation **viSetAttribute**. Although the INTFC resource does not have **viSetAttribute** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

The INTFC Resource lets a controller interact with any devices connected to the board associated with this resource. Services are provided to send blocks of data onto the bus, request blocks of data from the bus, trigger devices on the bus, and send miscellaneous commands to any or all devices. In addition, the controller can directly query and manipulate specific lines on the bus and also pass control to other devices with controller capability.

INTFC Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic INTFC Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_GPIB**	**VI_INTF_GPIB**
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_SEND_END_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_TRUE**

Attribute Name	Access Privilege		Data Type	Range	Default
Generic INTFC Resource Attributes (continued)					
VI_ATTR_TERMCHAR	RW	Local	**ViUInt8**	0 to FF_h	$0A_h$ (linefeed)
VI_ATTR_TERMCHAR_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
VI_ATTR_TMO_VALUE	RW	Local	**ViUInt32**	**VI_TMO_IMMEDIATE** 1 to $FFFFFFFE_h$ **VI_TMO_INFINITE**	2000 msec
VI_ATTR_DEV_STATUS_BYTE	RW	Global	**ViUInt8**	0 to FF_h	N/A
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	**ViUInt16**	**VI_FLUSH_ON_ACCESS** **VI_FLUSH_WHEN_FULL**	**VI_FLUSH_WHEN_FULL**
VI_ATTR_DMA_ALLOW_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	N/A
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	**ViUInt16**	**VI_FLUSH_ON_ACCESS** **VI_FLUSH_DISABLE**	**VI_FLUSH_DISABLE**
VI_ATTR_FILE_APPEND_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
GPIB Specific INTFC Resource Attributes					
VI_ATTR_GPIB_PRIMARY_ADDR	RW	Global	**ViUInt16**	0 to 30	N/A
VI_ATTR_GPIB_SECONDARY_ADDR	RW	Global	**ViUInt16**	0 to 31, **VI_NO_SEC_ADDR**	**VI_NO_SEC_ADDR**
VI_ATTR_GPIB_REN_STATE	RO	Global	**ViInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_GPIB_ATN_STATE	RO	Global	**ViInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_GPIB_NDAC_STATE	RO	Global	**ViInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A

Attribute Name	Access Privilege		Data Type	Range	Default
GPIB Specific INTFC Resource Attributes (continued)					
VI_ATTR_GPIB_SRQ_STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_GPIB_CIC_STATE	RO	Global	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_GPIB_SYS_CNTRL_STATE	RW	Global	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_GPIB_HS488_CBL_LEN	RW	Global	ViInt16	1 to 15, VI_GPIB_HS488_DISABLED, VI_GPIB_HS488_NIMPL	N/A
VI_ATTR_GPIB_ADDR_STATE	RO	Global	ViInt16	VI_GPIB_UNADDRESSED VI_GPIB_TALKER VI_GPIB_LISTENER	N/A

INTFC Resource Attribute Descriptions

Attribute Name	Description
Generic INTFC Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_SEND_END_EN	Whether to assert END during the transfer of the last byte of the buffer.
VI_ATTR_TERMCHAR	Termination character. When the termination character is read and **VI_ATTR_TERMCHAR_EN** is enabled during a read operation, the read operation terminates
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a termination character is received.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.

Attribute Name	Description
Generic INTFC Resource Attributes (continued)	
VI_ATTR_DEV_STATUS_BYTE	This attribute specifies the 488-style status byte of the local controller associated with this session. If this attribute is written and bit 6 (0x40) is set, this device or controller will assert a service request (SRQ) if it is defined for this interface.
VI_ATTR_WR_BUF_OPER_MODE	Determines the operational mode of the write buffer. When the operational mode is set to **VI_FLUSH_WHEN_FULL** (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up. If the operational mode is set to **VI_FLUSH_ON_ACCESS**, the write buffer is flushed under the same conditions, and also every time a **viPrintf** operation completes.
VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/O accesses should use DMA (**VI_TRUE**) or Programmed I/O (**VI_FALSE**). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode of the read buffer. When the operational mode is set to **VI_FLUSH_DISABLE** (default), the buffer is flushed only on explicit calls to **viFlush**. If the operational mode is set to **VI_FLUSH_ON_ACCESS**, the buffer is flushed every time a **viScanf** operation completes.
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether **viReadToFile** will overwrite (truncate) or append when opening a file.
GPIB Specific INTFC Resource Attributes	
VI_ATTR_GPIB_PRIMARY_ADDR	Primary address of the local GPIB controller used by the given session.
VI_ATTR_GPIB_SECONDARY_ADDR	Secondary address of the local GPIB controller used by the given session.
VI_ATTR_GPIB_REN_STATE	This attribute returns the current state of the GPIB REN (Remote ENable) interface line.
VI_ATTR_GPIB_ATN_STATE	This attribute shows the current state of the GPIB ATN (ATtentioN) interface line.
VI_ATTR_GPIB_NDAC_STATE	This attribute shows the current state of the GPIB NDAC (Not Data ACcepted) interface line.

Attribute Name	Description
GPIB Specific INTFC Resource Attributes (continued)	
`VI_ATTR_GPIB_SRQ_STATE`	This attribute shows the current state of the GPIB SRQ (Service ReQuest) interface line.
`VI_ATTR_GPIB_CIC_STATE`	This attribute shows whether the specified GPIB interface is currently CIC (controller in charge).
`VI_ATTR_GPIB_SYS_CNTRL_STATE`	This attribute shows whether the specified GPIB interface is currently the system controller. In some implementations, this attribute may be modified only through a configuration utility. On these systems, this attribute is read only (RO).
`VI_ATTR_GPIB_HS488_CBL_LEN`	This attribute specifies the total number of meters of GPIB cable used in the specified GPIB interface. If HS488 is not implemented, querying this attribute should return the value `VI_GPIB_HS488_NIMPL`. On these systems, trying to set this attribute value will return error `VI_ERROR_NSUP_ATTR_STATE`.
`VI_ATTR_GPIB_ADDR_STATE`	This attribute shows whether the specified GPIB interface is currently addressed to talk or listen, or is not addressed.

INTFC Resource Events

This resource defines the following events for communication with applications, where AP = Access Privilege.

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
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

Event Attribute	Description	AP	Data Type	Range
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 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

INTFC Resource Operations

viAssertTrigger(*vi, protocol*)
viBufRead(*vi, buf, count, retCount*)
viBufWrite(*vi, buf, count, retCount*)
viFlush(*vi, mask*)
viGpibCommand(*vi, buf, count, retCount*)
viGpibControlATN (*vi, mode*)
viGpibControlREN(*vi, mode*)
viGpibPassControl(*vi, primAddr, secAddr*)
viGpibSendIFC(*vi*)
viPrintf(*vi, writeFmt, arg1, arg2, ...*)
viRead(*vi, buf, count, retCount*)
viReadAsync(*vi, buf, count, jobId*)
viReadToFile(*vi, fileName, count, retCount*)
viScanf(*vi, readFmt, arg1, arg2, ...*)
viSetBuf(*vi, mask, size*)
viSPrintf(*vi, buf, writeFmt, arg1, arg2, ...*)
viSScanf(*vi, buf, readFmt, arg1, arg2, ...*)
viVPrintf(*vi, writeFmt, params*)
viVScanf(*vi, readFmt, params*)
viVSPrintf(*vi, buf, writeFmt, params*)
viVSScanf(*vi, buf, readFmt, params*)
viWrite(*vi, buf, count, retCount*)
viWriteAsync(*vi, buf, count, jobId*)
viWriteFromFile(*vi, fileName, count, retCount*)

VXI Mainframe Backplane (BACKPLANE) Resource

This section describes the VXI Mainframe Backplane (BACKPLANE) Resource that encapsulates the VXI-defined operations and properties of the backplane in a VXIbus system.

BACKPLANE Resource Overview

A VISA VXI Mainframe Backplane Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation **`viSetAttribute`**, which is defined in the VISA Resource Template.

Although the BACKPLANE resource does not have **`viSetAttribute`** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

The BACKPLANE Resource lets a controller query and manipulate specific lines on a specific mainframe in a given VXI system. Services are provided to map, unmap, assert, and receive hardware triggers, and also to assert various utility and interrupt signals. This includes advanced functionality that may not be available in all implementations or all vendors' controllers.

A VXI system with an embedded CPU with one mainframe will always have exactly one BACKPLANE resource. Valid examples of resource strings for this are **`VXI0::0::BACKPLANE`** and **`VXI::BACKPLANE`**. A multi-chassis VXI system may provide only one BACKPLANE resource total, but the recommended way is to provide one BACKPLANE resource per chassis, with the resource string address corresponding to the attribute **`VI_ATTR_MAINFRAME_LA`**. If a multi-chassis VXI system provides only one BACKPLANE resource, it is assumed to control the backplane resources in all chasses.

NOTE

Some VXI or GPIB-VXI implementations view all chasses in a VXI system as one entity. In these configurations, separate BACKPLANE resources are not possible.

BACKPLANE Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic BACKPLANE Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_VXI** **VI_INTF_GPIB_VXI**	N/A
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_TMO_VALUE	RW	Local	**ViUInt32**	**VI_TMO_IMMEDIATE** 1 to $FFFFFFFE_h$ **VI_TMO_INFINITE**	2000 msec
VXI and GPIB-VXI Specific BACKPLANE Resource Attributes					
VI_ATTR_TRIG_ID	RW	Local	**ViInt16**	**VI_TRIG_TTL0** to **VI_TRIG_TTL7**; **VI_TRIG_ECL0** to **VI_TRIG_ECL1**	N/A
VI_ATTR_MAINFRAME_LA	RO	Global	**ViInt16**	0 to 255 **VI_UNKNOWN_LA**	N/A
VI_ATTR_VXI_VME_SYSFAIL_STATE	RO	Global	**ViInt16**	**VI_STATE_ASSERTED** **VI_STATE_UNASSERTED** **VI_STATE_UNKNOWN**	N/A
VI_ATTR_VXI_VME_INTR_STATUS	RO	Global	**ViUInt16**	N/A	N/A
VI_ATTR_VXI_TRIG_STATUS	RO	Global	**ViUInt32**	N/A	N/A
VI_ATTR_VXI_TRIG_SUPPORT	RO	Global	**ViUInt32**	N/A	N/A

BACKPLANE Resource Attribute Descriptions

Attribute Name	Description
Generic BACKPLANE Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.
VXI and GPIB-VXI Specific BACKPLANE Resource Attributes	
VI_ATTR_TRIG_ID	Identifier for the current triggering mechanism.
VI_ATTR_MAINFRAME_LA	This is the logical address of a given device in the mainframe, usually the device with the lowest logical address. Other possible values include the logical address of the Slot 0 controller or of the parent-side extender. Often, these are all the same value. The purpose of this attribute is to provide a unique ID for each mainframe. A VISA manufacturer can choose any of these values, but must be consistent across mainframes. If this value is not known, the attribute value returned is **VI_UNKNOWN_LA**.
VI_ATTR_VXI_VME_SYSFAIL_STATE	This attribute shows the current state of the VXI/VME SYSFAIL (SYStem FAILure) backplane line.
VI_ATTR_VXI_VME_INTR_STATUS	This attribute shows the current state of the VXI/VME interrupt lines. This is a bit vector with bits 0-6 corresponding to interrupt lines 1-7.
VI_ATTR_VXI_TRIG_STATUS	This attribute shows the current state of the VXI trigger lines. This is a bit vector with bits 0-9 corresponding to **VI_TRIG_TTL0** through **VI_TRIG_ECL1**.
VI_ATTR_VXI_TRIG_SUPPORT	This attribute shows which VXI trigger lines this implementation supports. This is a bit vector with bits 0-9 corresponding to **VI_TRIG_TTL0** through **VI_TRIG_ECL1**. Agilent VISA also returns *12* to indicate **VI_TRIG_PANEL_IN** for received triggers and **VI_TRIG_PANEL_OUT** for asserted triggers on Agilent VXI controllers.

BACKPLANE Resource Events

This resource defines the following events for communication with applications, where AP = Access Privilege.

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`
`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.

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

BACKPLANE Resource Operations

viAssertTrigger(*vi, protocol*)
viMapTrigger(*vi, trigSrc, trigDest, mode*)
viUnmapTrigger(*vi, trigSrc, trigDest*)

Servant Device-Side (SERVANT) Resource

This section describes the Servant Device-Side (SERVANT) Resource that encapsulates the operations and properties of the capabilities of a device and a device's view of the system in which it exists.

NOTE

The SERVANT Resource is not implemented in Agilent VISA.

The SERVANT resource is a class for advanced users who want to write firmware code that exports device functionality across multiple interfaces. Most VISA users will not need this level of functionality and should not use the SERVANT resource in their applications.

A VISA user of the TCPIP SERVANT resource should be aware that each VISA session corresponds to a unique socket connection. If the user opens only one SERVANT session, this precludes multiple clients from accessing the device.

SERVANT Resource Overview

A VISA Servant Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation **`viSetAttribute`**, which is defined in the VISA Resource Template.

Although the SERVANT resource does not have **`viSetAttribute`** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

The SERVANT Resource exposes the device-side functionality of the device associated with this resource. Services are provided to receive blocks of data from a commander and respond with blocks of data in return, setting a 488-style status byte, and receiving device clear and trigger events.

SERVANT Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic SERVANT Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_VXI** **VI_INTF_GPIB** **VI_INTF_TCPIP**	N/A
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_SEND_END_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_TRUE**
VI_ATTR_TERMCHAR	RW	Local	**ViUInt8**	0 to FF_h	$0A_h$ (linefeed)
VI_ATTR_TERMCHAR_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
VI_ATTR_TMO_VALUE	RW	Local	**ViUInt32**	**VI_TMO_IMMEDIATE** 1 to $FFFFFFFE_h$ **VI_TMO_INFINITE**	2000 msec
VI_ATTR_DEV_STATUS_BYTE	RW	Local	**ViInt16**	0 to FF_h	N/A
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	**ViUInt16**	**VI_FLUSH_ON_ACCESS** **VI_FLUSH_WHEN_FULL**	**VI_FLUSH_WHEN_FULL**
VI_ATTR_DMA_ALLOW_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	N/A
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	**ViUInt16**	**VI_FLUSH_ON_ACCESS** **VI_FLUSH_DISABLE**	**VI_FLUSH_DISABLE**
VI_ATTR_FILE_APPEND_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_FALSE**
GPIB Specific SERVANT Resource Attributes					
VI_ATTR_GPIB_PRIMARY_ADDR	RO	Global	**ViUInt16**	0 to 30	N/A
VI_ATTR_GPIB_SECONDARY_ADDR	RO	Global	**ViUInt16**	0 to 31 **VI_NO_SEC_ADDR**	**VI_NO_SEC_ADDR**

Attribute Name	Access Privilege		Data Type	Range	Default
GPIB Specific SERVANT Resource Attributes (continued)					
VI_ATTR_GPIB_REN_STATE	RO	Global	**ViUInt16**	**VI_STATE_UNKNOWN** **VI_STATE_ASSERTED** **VI_STATE_UNASSERTED**	N/A
VI_ATTR_GPIB_ADDR_STATE	RO	Global	**ViUInt16**	**VI_GPIB_UNADDRESSED** **VI_GPIB_TALKER** **VI_GPIB_LISTENER**	N/A
VXI Specific SERVANT Resource Attributes					
VI_ATTR_VXI_LA	RO	Global	**ViUInt16**	0 to 511	N/A
VI_ATTR_CMDR_LA	RO	Global	**ViUInt16**	0 to 255, **VI_UNKNOWN_LA**	N/A
TCPIP Specific SERVANT Resource Attributes					
VI_ATTR_TCPIP_DEVICE_NAME	RO	Global	**ViString**	N/A	N/A

SERVANT Resource Attribute Descriptions

Attribute Name	Description
Generic SERVANT Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_SEND_END_EN	Whether to assert END during the transfer of the last byte of the buffer.
VI_ATTR_TERMCHAR	Termination character. When the termination character is read and **VI_ATTR_TERMCHAR_EN** is enabled during a read operation, the read operation terminates.
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a termination character is received.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.

Attribute Name	Description
Generic SERVANT Resource Attributes (continued)	
VI_ATTR_DEV_STATUS_BYTE	This attribute specifies the 488-style status byte of the local controller associated with this session.
VI_ATTR_WR_BUF_OPER_MODE	Determines the operational mode of the write buffer. When the operational mode is set to **VI_FLUSH_WHEN_FULL** (default), the buffer is flushed when an END indicator is written to the buffer or when the buffer fills up. If the operational mode is set to **VI_FLUSH_ON_ACCESS**, the write buffer is flushed under the same conditions, and also every time a **viPrintf** operation completes.
VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/O accesses should use DMA (**VI_TRUE**) or Programmed I/O (**VI_FALSE**). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode of the read buffer. When the operational mode is set to **VI_FLUSH_DISABLE** (default), the buffer is flushed only on explicit calls to **viFlush**.
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether **viReadToFile** will overwrite (truncate) or append when opening a file.
GPIB Specific SERVANT Resource Attributes	
VI_ATTR_GPIB_PRIMARY_ADDR	Primary address of local GPIB controller used by given session.
VI_ATTR_GPIB_SECONDARY_ADDR	Secondary address of the local GPIB controller used by the given session.
VI_ATTR_GPIB_REN_STATE	Returns the current state of the GPIB REN (Remote ENable) interface line.
VI_ATTR_GPIB_ADDR_STATE	Shows whether the specified GPIB interface is currently addressed to talk to listen, or to not addressed.
VXI Specific SERVANT Resource Attributes	
VI_ATTR_VXI_LA	Logical address of the VXI or VME device used by the given session. For a VME device, the logical address is actually a pseudo-address in the range 256 to 511.
VI_ATTR_CMDR_LA	Logical address of the commander of the VXI device used by the given session.

Attribute Name	Description
TCPIP Specific SERVANT Resource Attributes	
VI_ATTR_TCPIP_DEVICE_NAME	Specifies the LAN device name used by the VXI-11 protocol during connection.

SERVANT Resource Events

This resource defines the following events for communication with applications, where AP = Access Privilege.

VI_EVENT_CLEAR

Notification that the local 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_IO_COMPLETION

Notification that an asynchronous operation has completed.

Event Attributes	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 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

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_TRIG

Notification that the local controller has been triggered.

Event Attributes	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_SW

VI_EVENT_VXI_VME_SYSRESET

Notification that the VXI/VME SYSRESET* line has been reset.

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

VI_EVENT_TCPIP_CONNECT

Notification that a TCPIP connection has been made.

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_RECV_ TCPIP_ADDR	The TCPIP address of the device from which the session received a connection.	RO	**ViString**	N/A

SERVANT Resource Operations

viBufRead(*vi, buf, count, retCount*)
viBufWrite(*vi, buf, count, retCount*)
viFlush(*vi, mask*)
viPrintf(*vi, writeFmt, arg1, arg2, ...*)
viRead (*vi, buf, count, retCount*)

viReadAsync(*vi, buf, count, jobId*)
viReadToFile(*vi, fileName, count, retCount*)
viScanf(*vi, readFmt, arg1, arg2, ...*)
viSetBuf(*vi, mask, size*)
viSPrintf(*vi, buf, writeFmt, arg1, arg2, ...*)

viSScanf(*vi, buf, readFmt, arg1, arg2, ...*)
viVPrintf(*vi, writeFmt, params*)
viVScanf(*vi, readFmt, params*)
viVSPrintf(*vi, buf, writeFmt, params*)
viVSScanf(*vi, buf, readFmt, params*)

viWrite(*vi, buf, count, retCount*)
viWriteAsync(*vi, buf, count, jobId*)
viWriteFromFile(*vi, fileName, count, retCount*)

TCPIP Socket (SOCKET) Resource

This section describes the TCPIP Socket (SOCKET) Resource that encapsulates the operations and properties of the capabilities of a raw network socket connection using TCPIP.

SOCKET Resource Overview

A VISA SOCKET Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation **`viSetAttribute`**, which is defined in the VISA Resource Template.

Although the TCPIP resource does not have **`viSetAttribute`** listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

The SOCKET Resource exposes the capability of a raw network socket connection over TCPIP. This ususally means Ethernet, but the protocol is not restricted to that physical interface. Services are provided to send and receive blocks of data. If the device is capable of communicating with 488.2-style strings, an attribute setting also allows sending software triggers, querying a 488-style status byte, and sending a device clear message.

SOCKET Resource Attributes

Attribute Name	Access Privilege		Data Type	Range	Default
Generic SOCKET Resource Attributes					
VI_ATTR_INTF_NUM	RO	Global	**ViUInt16**	0 to $FFFF_h$	0
VI_ATTR_INTF_TYPE	RO	Global	**ViUInt16**	**VI_INTF_TCPIP**	**VI_INTF_TCPIP**
VI_ATTR_INTF_INST_NAME	RO	Global	**ViString**	N/A	N/A
VI_ATTR_SEND_END_EN	RW	Local	**ViBoolean**	**VI_TRUE** **VI_FALSE**	**VI_TRUE**

Attribute Name	Access Privilege		Data Type	Range	Default
Generic SOCKET Resource Attributes (continued)					
VI_ATTR_TERMCHAR	RW	Local	ViUInt8	0 to FF_h	$0A_h$ (linefeed)
VI_ATTR_TERMCHAR_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_TMO_VALUE	RW	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to $FFFFFFFE_h$ VI_TMO_INFINITE	2000 msec
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL	VI_FLUSH_WHEN_FULL
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_DISABLE	VI_FLUSH_DISABLE
VI_ATTR_FILE_APPEND_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_IO_PROT	RW	Local	ViUInt16	VI_NORMAL VI_PROT_4882_STRS	VI_NORMAL
TCPIP Specific SOCKET Resource Attributes					
VI_ATTR_TCPIP_ADDR	RO	Global	ViString	N/A	N/A
VI_ATTR_TCPIP_HOSTNAME	RO	Global	ViString	N/A	N/A
VI_ATTR_TCPIP_PORT	RO	Global	ViUInt16	0 to $FFFF_h$	N/A
VI_ATTR_TCPIP_NODELAY	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_TCPIP_KEEPALIVE	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE

SOCKET Resource Attribute Descriptions

Attribute Name	Description
Generic SOCKET Resource Attributes	
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_SEND_END_EN	Whether to assert END during the transfer of the last byte of the buffer.
VI_ATTR_TERMCHAR	Termination character. When the termination character is read and **VI_ATTR_TERMCHAR_EN** is enabled during a read operation, the read operation terminates.
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a termination character is received.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of **VI_TMO_IMMEDIATE** means that operations should never wait for the device to respond. A timeout value of **VI_TMO_INFINITE** disables the timeout mechanism.
VI_ATTR_WR_BUF_OPER_MODE	Determines the operational mode of the write buffer. When the operational mode is set to **VI_FLUSH_WHEN_FULL** (default), the buffer is flushed when an END indicator is written to the buffer or when the buffer fills up. If the operational mode is set to **VI_FLUSH_ON_ACCESS**, the write buffer is flushed under the same conditions, and also every time a **viPrintf** operation completes.
VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/O accesses should use DMA (**VI_TRUE**) or Programmed I/O (**VI_FALSE**). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode of the read buffer. When the operational mode is set to **VI_FLUSH_DISABLE** (default), the buffer is flushed only on explicit calls to **viFlush**.
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether **viReadToFile** will overwrite (truncate) or append when opening a file.
VI_ATTR_IO_PROT	Specifies which protocol to use.

Attribute Name	Description
TCPIP SOCKET Resource Attributes	
VI_ATTR_TCPIP_ADDR	This is the TCPIP address of the device to which the session is connected. This string is formatted in dot notation.
VI_ATTR_TCPIP_HOSTNAME	Specifies the host name of the device. If no host name is available, this attribute returns an empty string.
VI_ATTR_TCPIP_PORT	Specifies the port number for a given TCPIP address. For a TCPIP SOCKET resource, this is a required part of the address string.
VI_ATTR_TCPIP_NODELAY	The Nagle algorithm is disabled when this attribute is enabled (and vice versa). The Nagle algorithm improves network performance by buffering "send" data until a full-size packet can be sent. This attribute is enabled by default in VISA to verify that synchronous writes get flushed immediately.
VI_ATTR_TCPIP_KEEPALIVE	An application can request that a TCPIP provider enable the use of "keep-alive" packets on TCP connections by turning on this attribute. If a connection is dropped as a result of "keep-alives," the error code **VI_ERROR_CONN_LOST** is returned to current and subsequent I/O calls on the session.

SOCKET Resource Event

This resource defines the following events for communication with applications, where AP = Access Privilege.

VI_EVENT_IO_COMPLETION

Notification that an asynchronous operation has completed.

Event Attributes	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

Event Attributes	Description	AP	Data Type	Range
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

SOCKET Resource Operations

viAssertTrigger(*vi, protocol*)
viBufRead(*vi, buf, count, retCount*)
viBufWrite(*vi, buf, count, retCount*)
viClear(*vi*)
viFlush(*vi, mask*)

viPrintf(*vi, writeFmt, arg1, arg2, ...*)
viRead(*vi, buf, count, retCount*)
viReadAsync(*vi, buf, count, jobId*)
viReadSTB(*vi, status*)
viReadToFile(*vi, filename, count, retCount*)

viScanf(*vi, readFmt, arg1, arg2, ...*)
viSetBuf(*vi, mask, size*)
viSPrintf(*vi, buf, writeFmt, arg1, arg2, ...*)
viSScanf(*vi, buf, readFmt, arg1, arg2, ...*)
viVPrintf(*vi, writeFmt, params*)

viVScanf(*vi, readFmt, params*)
viVSPrintf(*vi, buf, writeFmt, params*)
viVSScanf(*vi, buf, readFmt, params*)
viWrite(*vi, buf, count, retCount*)
viWriteAsync(*vi, buf, count, jobId*)
viWriteFromFile(*vi, filename, count, retCount*)

Glossary

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 driver
A segment of software code that communicates with a device. It may either communicate directly with a device by reading to and writing from registers, or it may communicate through an interface driver.

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.

VISA
Virtual Instrument Software Architecture. VISA is a common I/O library where software from different vendors can run together on the same platform.

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. This is 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 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

E (continued)

F

F (continued)

F (continued)

G

H

I

T

U

V

V (continued)

V (continued)

W