

Welcome to the Agilent VISA Help

VISA (Virtual Instrument Software Architecture) is an industrystandard I/O API. It can be used to develop I/O applications and

instrument drivers that are interoperable with many other VISA applications from many vendors, and that comply with <u>www.ivifoundation.org</u> standards. VXIplug&play drivers are implemented using VISA.

The reference information for Agilent VISA is organized into the following major sections.

VISA User's Guide

Describes the Agilent VISA library and shows how to use it to develop I/O applications and instrument drivers on Windows ® PCs.

VISA Function Reference

Lists the Agilent VISA functions and provides brief descriptions of their usage.

VISA Attributes

Describes attributes for the VISA template and supported Agilent VISA Resource Classes, including INSTR, MEMACC, INTFC, BACKPLANE, and SOCKET.

VISA Library Information

Describes <u>VISA type definitions</u>, <u>VISA error codes</u>, and <u>VISA directories</u>, with additional information on using VISA in <u>Visual Basic</u> and in <u>Microsoft</u>.<u>.NET languages</u>.

For general VISA specifications, see the *VXIplug&play Systems Alliance VISA Library* specifications.

Agilent VISA provides support for version 5.0 of the VISA specification which is available from the <u>www.ivifoundation.org</u>.

What's in This Guide?

This user's guide shows VISA programming techniques using C/C++ and Visual Basic. For information on .NET programming languages, refer to <u>VISA .NET</u> <u>Tutorial</u>.

This guide contains these sections:

- <u>VISA Overview</u> Provides a short description of the Agilent Virtual Instrument Software Architecture (VISA).
- <u>Programming with VISA</u> Describes the basics of VISA and lists some sample programs. The section also includes information on how to create sessions, how to use formatted I/O, and how to use events.
- <u>Programming via GPIB and VXI</u> Describes how to use VISA to communicate over the GPIB, GPIB-VXI, and VXI interfaces to instruments.
- <u>Programming PXI Devices</u> Describes how to use VISA to program PXI, PXIe, and PCIe devices installed in a PC or PXI chassis.
- <u>Programming via LAN</u> Describes how to use VISA to communicate over a LAN (Local Area Network) to instruments.
- <u>Programming via USB</u> Describes how to use VISA to communicate over a USB (Universal Serial Bus) to instruments.

See Also

VISA Documentation lists other sources of information on VISA programming.

VISA Overview

VISA is an application programming interface (API) for instrument control. It allows you to programmatically send commands and receive data from instruments and other test and measurement devices (such as sources and switches).

VISA is a part of the Agilent IO Libraries Suite product. The Agilent IO Libraries Suite includes three VISA-related APIs:

- Agilent Virtual Instrument Software Architecture (VISA)
- VISA for the Common Object Model (VISA COM)
- Agilent Standard Instrument Control Library (SICL)

For information on any API, click the Agilent IO Control [®] in your Windows taskbar, then click **Documentation** > **API Documentation** and select a document from the displayed list.

Using VISA, VISA COM, and SICL

Agilent Virtual Instrument Software Architecture (VISA) is an I/O library that allows software developed from different vendors to run on the same system.

If you are using new instruments or are developing new I/O applications or instrument drivers, and you have chosen to use direct I/O rather than instrument drivers, we recommend you use Agilent VISA or VISA COM. See the Agilent IO Libraries Suite Help for an in-depth discussion of your programming options.

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 supported on the GPIB, VXI, GPIB-VXI, Serial (RS-232), LAN, PXI, and USB interfaces. LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

Agilent VISA provides support for version 5.0 of the VISA specification which is available from <u>www.ivifoundation.org</u>.

VISA Documentation

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

Document	Description
Agilent IO Libraries Suite Connectivity Guide	 This guide brings together all of the Agilent IO Libraries Suite documents and related Web sites into one help document. This guide provides easy access to: Agilent interface manuals The Agilent IO Libraries Quick Start Guide Connection Expert, VISA, VISA COM, LXI, and Agilent 488 help An introductory video Related Web sites
VISA Help	(This help file.) Contains a VISA user's guide, a function reference, and other programming information.
VISA Sample Programs	Sample programs are provided online to help you develop VISA applications. See <u>Example Programs and Installation Folders</u> for locations.
VISA Specification	Specifications for VISA: <u>www.ivifoundation.org</u> .
PXI Specification	Specifications for PXI: <u>www.pxisa.org</u> .
IEEE Standard Codes, Formats, Protocols, and Common Commands	ANSI/IEEE Standard 488.2-1992: <u>www.standards.ieee.org</u> .
	TCP/IP Instrument Protocol Specification - VXI-11, Rev. 1.0
	TCP/IP-VXIbus Interface Specification - VXI-11.1, Rev. 1.0
VXI Bus Consortium Specifications (when using VISA over LAN)	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
	These specifications are available from the VXI Bus Consortium at: <u>www.vxi.org</u>

Programming with VISA

This section describes how to program with VISA. The basics of VISA are described, including formatted I/O, events and handlers, attributes, and locking. Topics are:

- Example Programs and Installation Folders
- <u>VISA Resources and Attributes</u>
- Using Sessions
- Sending I/O Commands
- Using Events and Handlers
- <u>Trapping Errors</u>
- Logging Error Messages
- Using Locks

Example Programs and Installation Folders

Most example programs listed in this help are installed by default in:

C:*Documents and Settings**All Users**Agilent**IO Libraries Suite Programming Samples* (on Windows XP)

C:\ProgramData\Agilent\Agilent IO Libraries Programming Samples (on Windows Vista, Windows 7, Windows 8, Windows Server 2008 R2, or Windows Server 2012)

Click the Agilent IO Control @ and select **About Agilent IO Control** to see the specific installation directories used on your PC, as shown below.

VISA Resources and Attributes

This section introduces VISA resources and 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 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.

Note: Although the Servant Device-Side (SERVANT) resource is defined by the VISA specification, the SERVANT resource is not supported by Agilent VISA. The SERVANT resource is intended for advanced users who need to write code that causes a computer to function as an instrument rather than as a controller.

The table below describes each resource class supported by Agilent VISA.

Resource Class	Interface Types	Resource Class Description
<u>Instrument Control (INSTR)</u> <u>Resource</u>	Generic, GPIB, GPIB-VXI, PXI, Serial, TCPIP, USB, VXI	Device operations (reading, writing, triggering, etc.).
<u>GPIB Bus Interface (INTFC)</u> <u>Resource</u>	Generic, GPIB	Raw GPIB interface operations (reading, writing, triggering, etc.).
Memory Access (MEMACC) Resource	Generic, GPIB-VXI, PXI, VXI	Address space of a memory-mapped bus, such as the VXIbus.
VXI Mainframe Backplane (BACKPLANE) Resource	Generic, GPIB-VXI, VXI (GPIB-VXI BACKPLANE not supported)	VXI-defined operations and properties of each backplane in a VXIbus system.
<u>TCPIP Socket (SOCKET)</u> <u>Resource</u>	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, declare a variable of that type and use it for the returned data. If ViString is returned, allocate an array and pass a pointer to that array for returned data.

Example: Reading a VISA Attribute

This example reads the state of the VI_ATTR_TERMCHAR_EN attribute and, if it is false, changes the state to 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:

- Opening a Session
- <u>Addressing a Session</u>
- <u>Closing a Session</u>
- 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.

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 return a unique session to the same default resource manager resource.

Resource Sessions

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

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

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

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

sesn	A session returned from the viOpenDefaultRM function that identifies the resource manager session.
rsrcName	A unique symbolic name of the resource (resource address).
accessMode	Specifies the modes by which the resource is to be accessed. The value VI_EXCLUSIVE_LOCK is used to acquire an exclusive lock immediately upon opening a session. If a lock cannot be acquired, the session is closed and an error is returned. The VI_LOAD_CONFIG value is used to configure attributes specified by some external configuration utility. If this value is not used, the session uses the default values provided by this specification.
	Multiple access modes can be used simultaneously by specifying a "bit-wise OR" of the values.
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.
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.

Parameter Descriptions

Example: Opening a Resource Session

This code sample shows one way of opening resource sessions with a GPIB multimeter and a GPIB-VXI scanner. The sample first opens a session with the default resource manager. The example then uses the session returned from the resource manager, and a VISA address, to open a session with the GPIB device at address 22. You can now identify that session as dmm when you call other VISA functions.

The example uses the session returned from the resource manager, with another VISA address, to open a session with the GPIB-VXI device at primary address 9 and VXI logical address (secondary address) 24. You can then identify this session as *scanner* when calling other VISA functions. See <u>Addressing a Session</u>, 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 ID and the resource address. The interface ID is determined when you run the Agilent Connection Expert utility. The interface ID is usually the VISA interface type followed by a number.

The following table illustrates the format of the rsrcName for different VISA 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 used for asynchronous serial devices.
- GPIB used for GPIB devices and interfaces.
- GPIB-VXI used for GPIB-VXI controllers.
- PXI used for modular instruments.
- TCPIP used for LAN and HiSLIP instruments.
- VXI used for VXI instruments.
- USB used for USB 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
PXI	PXI[bus]::device[::function][::INSTR]
PXI	PXI[interface]::bus-device[.function][::INSTR]
PXI	PXI[interface]::CHASSISchassis::SLOTslot[::FUNCfunction][::INSTR]
PXI	PXI[interface]::MEMACC
TCPIP	TCPIP[board]::host address[::LAN device name]::INSTR
TCPIP	TCPIP[board]::host address[::HiSLIP device name[,HiSLIP port]][::INSTR]

TCPIP	TCPIP[board]::host address::port::SOCKET
USB	USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]
VXI	VXI[board]::VXI logical address[::INSTR]
VXI	VXI[board]::MEMACC
VXI	VXI[board][::VXI logical address]::BACKPLANE

The following table describes the parameters shown in the table 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	A hostname or a dot-delimited IPv4 IP address for TCPIP. The IP address (in dot- delimited decimal notation) or the name of the host computer/gateway. For the HiSLIP and Raw Sockets protocols only, the host address can be an IPv6 address in square brackets, [fe80::218:e77f] for example.
HiSLIP device name	The assigned name for the HiSLIP device (the name must begin with hislip). Only address strings specifically requesting a HiSLIP connection via the HiSLIP device name will get a HiSLIP connection. The default HiSLIP device name is hislip0.
HiSLIP port	The port number to use for a HiSLIP connection. The default is 4880.
LAN device name	The assigned name for a LAN device. Device name depends on LAN Host Device (see documentation that came with your LAN Host device, such as E5810A). The default is inst0.
manufacturer ID	Manufacturer's ID for a USB Test & Measurement-class device
model code	Model code of a USB device.
port	The port number to use for a TCP/IP socket connection.
primary address	The primary address of the GPIB device.
secondary address	This optional parameter is the secondary address of the GPIB device. If no secondary address is specified, none is assumed.
serial number	Serial number of a USB device.
USB interface number	Interface number of a USB device.
VXI logical address	Logical address of a VXI instrument within a mainframe.

Some examples of valid VISA addresses 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.
PXI0::3-18::INSTR	PXI device 18 on bus 3.
PXI0::3-18.2::INSTR	Function 2 on PXI device 18 on bus 3.
PXI0::21::INSTR	PXI device 21 on bus 0.
PXI0::CHASSIS1::SLOT4::INSTR	PXI device in slot 4 of chassis 1.
PXI0::MEMACC	Access to system controller memory available to PXI devices.
TCPIP0::1.2.3.4::999::SOCKET	Raw TCPIP access to port 999 at the specified address.
TCPIP0::[fe80::1]::999::SOCKET	Raw TCPIP using IPv6 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.
TCPIP0::[fe80::1]::hislip0::INSTR	A TCP/IP device using HiSLIP located at IPv6 IP address fe80::1.
USB::0x1234::0x5678::A22-5::INSTR	A USB device with manufacturer ID 0x1234, model code 0x5678, and serial number A22-5. This uses the device's first available USBTMC interface. This is usually number 0.

Example: Opening a Session

This sample shows one way to open a VISA 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

You must use the viClose function to close each session. Closing the specific resource session frees all data structures that have been allocated for the session. If you close the default resource manager session, all sessions opened using that resource manager session will close.

Since system resources are also used when searching for resources (viFindRsrc), the viClose function needs to be called to free up find lists. See <u>Searching for</u> <u>Resources</u> for more information on closing find lists.

Searching for Resources

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

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

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

The parameters are defined as follows.

Parameter Description

sesn	The resource manager session.
expr	The expression that identifies what to search (see table below).
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 viFindRsrc 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.

expr Parameter
CPIR[0-9]*··?*INSTR
VXI?*INSTR
GPIB-VXI?*INSTR
GPIB?*INSTR
?*VXI[0-9]*::?*INSTR
ASRL[0-9]*::?*INSTR
?*INSTR

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

Example: Searching the VXI Interface for Resources

This code sample 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 topic contains guidelines for sending I/O commands, including:

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

Types of I/O

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

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

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

The VISA formatted I/O mechanism is similar to the C stdio mechanism. The VISA formatted I/O functions are viPrintf, viQueryf, and viScanf. Two nonbuffered and non-formatted I/O functions, viRead and viWrite, synchronously transfer data. Two additional functions, viReadAsync and viWriteAsync, asynchronously transfer data. These are raw I/O functions and do not intermix with the formatted I/O functions. See <u>Using Non-Formatted I/O</u> for details.

Using Formatted I/O

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

viPrintf formats data according to the format string (writeFmt) and sends the data to a device. viPrintf sends separate arg parameters, while the viVPrintf function sends a list of parameters in params:

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

viScanf receives and converts data from a device according to the format string (readFmt). 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);
```

viQueryf formats and sends data to a device and then immediately receives and converts the response data. Hence, viQueryf is a combination of both viPrintf and viScanf functions. Similarly, viVQueryf is a combination of viVPrintf and viVScanf. viQueryf 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

Formatted I/O functions convert data under the control of the format specifier. The format specifier consists of a % (percent) symbol, optional modifier, and a format code. Both readFmt and writeFmt have the form:

%[modifier]formatCode

Example: Receiving Data From a Session

The following example uses viScanf to receive 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 <u>viSetBuf</u> 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 new line 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 <u>VISA Attributes</u> 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 <u>VISA Attributes</u> 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

The following C sample program demonstrates sending and receiving formatted I/O. The program opens a session with a GPIB device and sends a comma operator to send a comma-separated list. This program shows specific VISA functionality and does not include error trapping.

The <u>formatio.c sample program</u> is installed on your system in the *ProgrammingSamples* subdirectory.

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 <u>viRead</u>, <u>viWrite</u>, <u>viReadAsync</u>, <u>viWriteAsync</u>, or <u>viTerminate</u>.

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);

Example: Using Non-Formatted I/O Functions

The <u>nonfmtio.c sample program</u> illustrates using non-formatted I/O functions to communicate with a GPIB device. It 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 <u>Trapping Errors</u> for more information.

/*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 ("Mesurement Results: %lf\n", res);
```

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

/*nonfmtio.c

This example program measures the AC voltage on a multimeter and prints the results. Note that you must 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 ("Mesurement Results: %s\n", strres);

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

Using Events and Handlers

This section describes how to uses 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 call back 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 vilnstallHandler, 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 call back 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

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.

The events that are implemented for Agilent VISA for each resource class are:

- Instrument Control (INSTR) Resource Events
- <u>Memory Access (MEMACC) Resource Event</u>
- GPIB Bus Interface (INTFC) Resource Events
- <u>TCPIP Socket (SOCKET) Resource Event</u>

Note: Some resource classes/events, such as the SERVANT class, are not implemented by Agilent VISA and are not described in this help.

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 <u>VISA</u> <u>Attributes</u> for more information on reading attribute states.

ViInt16 state;

viGetAttribute(context, VI_ATTR_RECV_TRIG_ID, &state)

See Also

- Using the Callback Method
 Using the Queuing Method

Instrument Control (INSTR) Resource Events

In the following table, AP = Access Privilege, RO - Read Only, and RW = Read/Write.

VI_EVENT_SERVICE_REQUEST

Notification that a service request was received from the device.

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

VI_EVENT_VXI_SIGP

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI
VI_ATTR_SIGP_STATUS_ID	The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.	RO	ViUInt16	0 10 FFFFh

VI_EVENT_TRIG

Notification that a trigger interrupt was received from the device. For VISA, the only triggers that can be sensed are VXI h 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_TRI(
VI_ATTR_RECV_TRIG_ID	The identifier of the triggering mechanism on which the specified	RO	ViInt16	VI_TRIG_TTL0 t VI_TRIG_TTL7; VI_TRIG_ECL0 t

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

VI_EVENT_IO_COMPLETION

Notification that an asynchronous operation has completed.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_(
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 FFFFFFFFh
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A
VI_ATTR_RET_COUNT_32	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to FFFFFFFF _h
VI_ATTR_RET_COUNT_64*	Actual number of elements that were asynchronously transferred.	RO	ViUInt64	0 to FFFFFFFFFF

*Defined only for frameworks that are 64-bit native.

VI_EVENT_USB_INTR

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_USB
VI_ATTR_USB_RECV_INTR_SIZE	Specifies the size of the data that was received from the USB interrupt-IN pipe. This value will never be larger than the sessions value of VI_ATTR_ USB_MAX_INTR_SIZE.	RO	ViUInt16	0 to FFFF _h

VI_ATTR_USB_RECV_INTR_DATA	Specifies the actual data that was received from the USB interrupt-IN pipe. Querying this attribute copies the contents of the data to the users buffer. The users buffer must be sufficiently large enough to hold all of the data.	RO	ViBuf	N/A
VI_ATTR_STATUS	Specifies the status of the read operation from the USB interrupt-IN pipe. If the device sent more data than the user specified in VI_ATTR_USB_MAX_INTR_SIZE, then this attribute value will contain an error code.	RO	ViStatus	N/A

VI_EVENT_PXI_INTR

Event Attribute	vent Attribute Description		AP Data Type Range	
VI_ATTR_EVENT_TYPE	Notification that a PCI Interrupt was received from the device.	RO	ViEventType	VI_EVENT_PXI_

is text and replace it with your own content.

Memory Access (MEMACC) Resource Event

In the following table, AP = Access Privilege, RO - Read Only, and RW = Read/Write.

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

GPIB Bus Interface (INTFC) Resource Events

In the following table, AP = Access Privilege, RO - Read Only, and RW = Read/Write.

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	RO	ViBuf	N/A

	asynchronous operation.			
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to FFFFFFFh
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.

VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_ SYSFAIL

VI_EVENT_VXI_VME_SYSRESET

Notification that the VXI/VME SYSRESET* line has been reset.

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_ SYSRESET

TCPIP Socket (SOCKET) Resource Event

In the following table, AP = Access Privilege, RO - Read Only, and RW = Read/Write.

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

Using the Callback Method

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

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

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

Example: Using the Callback Method

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

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

/* your event handling code here */

```
return VI_SUCCESS;
```

```
}
main(){
ViSession vi;
ViAddr addr=0;
```

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

```
/* your code here */
```

```
viDisableEvent(vi, VI_EVENT_SERVICE_REQ, VI_HNDLR);
viUninstallHandler(vi, VI_EVENT_SERVICE_REQ, my_handler, addr);
```

```
.
}
```

Installing Handlers

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

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

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

These parameters are defined as follows.

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 <u>viUninstallHandler</u> 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 topic.

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 <u>Using the Queuing Method</u> for more information.

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 sample 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 <u>Trapping Errors</u> for more information.

The <u>evnthdlr.c sample program</u> illustrates installing an event handler to be called when a trigger interrupt occurs and is installed on your system in the *ProgrammingSamples* subdirectory.

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 sample 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 <u>Trapping Errors</u> for more information.

The <u>srqhdlr.c sample program</u> illustrates installing an event handler to be called when an SRQ interrupt occurs and is installed on your system in the *ProgrammingSamples* subdirectory. .

See Also

- Events and Attributes
- Using the Queuing Method

/* evnthdlr.c

This example program illustrates installing an event handler to be called when a trigger interrupt occurs. Note that you must 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);
```

}

/* srqhdlr.c

This example program illustrates installing an event handler to be called when an SRQ interrupt occurs. Note that you must 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,</pre>
```

```
(ViAddr)10);
```

```
/* Clean up after ourselves - don't leave device in error state */
viPrintf( vi, "*CLS\n" );
/* close the sessions */
viClose(vi);
viClose(defaultRM);
```

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

Using the Queuing Method

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

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

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

Example: Using the Queuing Method

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

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

Enabling Events

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

viEnableEvent(vi, eventType, mechanism, context);

These parameters are defined as follows:

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

The <u>evntqueu.c sample program</u> illustrates enabling an event queue using viWaitOnEvent and is installed on your system in the *ProgrammingSamples* subdirectory. See <u>Example Programs and Installation Folders</u> for locations of example programs. 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 <u>Trapping Errors</u> for more information.

See Also

- Events and Attributes
- Using the Callback Method

/* evntqueu.c

This example program illustrates enabling an event queue using viWaitOnEvent. Note that you must change the 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);
```

```
/* unenable event */
viDisableEvent(vi, VI_EVENT_TRIG, VI_QUEUE);
```

```
/* close the sessions */
```

```
viClose(vi);
viClose(defaultRM);
}
```

Trapping Errors

This topic describes how to trap errors and handle 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);</pre>

Example: Checking for VI_SUCCESS

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

ViStatus err;

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

Example: Printing Error Code

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

void err_handler(ViSession vi, ViStatus err){

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

}

Example: Checking Instrument Errors

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

```
void system_err(){
 ViStatus err;
 char buf[1024] = \{0\};
 int err no;
 err=viPrintf(vi, "SYSTEM:ERR?\n");
 if (err < VI_SUCCESS) err_handler (vi, err);
 err=viScanf (vi, "%d%t", &err_no, &buf);
 if (err < VI_SUCCESS) err_handler (vi, err);
 while (err_no >0){
  printf ("Error Found: %d,%s\n", err_no, buf);
  err=viScanf (vi, "%d%t", &err no, &buf);
 }
 err=viFlush(vi, VI READ BUF);
 if (err < VI_SUCCESS) err_handler (vi, err);
 err=viFlush(vi, VI_WRITE_BUF);
 if (err < VI_SUCCESS) err_handler (vi, err);
}
```

Exception Events

An alternative to trapping VISA errors by checking the return status after each VISA call is to use the VISA exception event. On sessions where an exception event handler is installed and VI_EVENT_EXCEPTION is enabled, the exception event handler is called whenever an error occurs while executing an operation.

Exception Handling Model

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

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

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

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

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

mechanisms, the control will not return to the VISA system. This has several important repercussions:

- 1. If multiple handlers were installed on the exception event, the handlers that were not invoked prior to the current handler will not be invoked for the current exception.
- 2. The exception context will not be deleted by the VISA system when a C++ exception is used. In this case, the application should delete the exception context as soon as the application has no more use for the context, before terminating the session. An application should use the viClose operation to delete the exception context.
- 3. Code in any operation (after calling an exception handler) may not be called if the handler does not return. For example, local allocations must be freed before invoking the exception handler, rather than after it.

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

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

Using the VI_EVENT_EXCEPTION Event

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

Attribute Name	Privilege		Туре	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 ) {</pre>
 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
);
}
```

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 Event Viewer utility or the Debug Window. There are three choices for VISA logging:

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

Using the Event Viewer

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; they indicate events which are being handled internally by VISA. The process for using the Event Viewer is:

- Enable VISA logging by clicking the Agilent IO Control In the taskbar and then clicking
 Agilent VISA Options > VISA Logging > Event Viewer.
- Run your VISA program.
- View VISA error messages by running the Event Viewer. From the Agilent IO Control ⁽¹⁾, click **Event Viewer**. VISA error messages will appear in the application log of the Event Viewer utility.

Using the Debug Window

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

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

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.

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.

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 attributes.

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

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

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

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

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.

Example: Exclusive Lock

The <u>lockexcl.c sample program</u> 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.

Example: Shared Lock

The <u>lockshr.c sample program</u> 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.

/* lockexcl.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");
```

/* Make sure no other process or thread does anything to this resource between the viPrintf() and the 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);
}
```

/* 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 too */
viUnlock (vi);
```

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

Programming via GPIB and VXI

VISA supports three interfaces you can use to access GPIB (General Purpose Interface Bus) and VXI (VME eXtension for Instrumentation) instruments: GPIB, VXI, and GPIB-VXI. This section describes how to program GPIB and VXI devices via the GPIB, VXI or GPIB-VXI interfaces, including:

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

See <u>Programming with VISA</u> for general information on VISA programming for the GPIB, VXI, and GPIB-VXI interfaces.

GPIB and VXI Interfaces Overview

This topic is an overview of the GPIB, GPIB-VXI, and VXI interfaces, including:

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

General Interface Information

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

What is an I/O Interface?

An I/O interface can be defined as both a hardware interface and a software interface. Connection Expert is used to associate a unique interface name with a hardware interface. Agilent IO Libraries Suite uses a VISA interface name to identify an interface. This information is passed in the parameter string of the viOpen function call in a VISA program.

Connection Expert assigns a VISA interface name to the interface hardware, and other necessary configuration values for an interface when the interface is configured. See the Agilent IO Libraries Suite Help for details.

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 nonformatted 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) installed in the PC, and one or more GPIB instruments connected to the GPIB cards via GPIB cable. I/O communication between the PC and the instruments is via the GPIB cards and the GPIB cable. The following figure shows GPIB instruments at addresses 3 and 5.



Example: GPIB (82350) Interface

The GPIB interface system in the following figure consists of a Windows PC with two 82350 GPIB cards connected to three GPIB instruments via GPIB cables. For this system, Agilent Connection Expert 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 below.

GPIB Interface (82350 PCI GPIB Cards)



VISA Addressing

viOpen ("GPIB0::5::INSTR")	Open IO path to GPIB instrument at address 5 using 82350	Card #1
viOpen ("GPIB0::3::INSTR")	Open IO path to GPIB instrument at address 3 using 82350	Card #1
viOpen ("GPIB1::3::INSTR")	Open IO path to GPIB instrument at address 3 using 82350	Card #2

VXI Interfaces Overview

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



Example: VXI (E8491B) Interfaces

The VXI interface system in the following figure consists of a Windows PC with an E8491 PC card that connects to an E8491B IEEE-1394 to VXI Module in a VXI Mainframe. For this system, the three VXI instruments shown have logical addresses 8, 16, and 24. The Connection Expert 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 (E8491B IEEE-1394 to VXI Module)



VISA Addressing

viOpen (... "VXI0::24::INSTR"...) Open IO path to VXI instrument at logical address 24

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 Command Modules is installed by the Agilent IO Libraries Suite 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 Agilent 82350 GPIB card that connects to an E1406A command module in a VXI mainframe. The VXI mainframe includes one or more VXI instruments.

When Agilent IO Libraries Suite was 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 Connection Expert 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 instrument's user's guide.



VISA Addressing

viOpen (... "GPIB-VXI0::24::INSTR"...) Open IO path to VXI instrument at logical address 24

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 trade-off, 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 <u>Using Low-Level Memory Functions</u>.

Programming the Registers

High-level memory functions include the viIn and viOut functions for transferring 8-, 16-, 32-, or 64-bit values, as well as the viMoveIn and viMoveOut functions for transferring 8-, 16-, 32-, or 64-bit blocks of data into or out of local memory. You can therefore program using 8-, 16-, 32-, or 64-bit transfers.

High-Level Memory Functions

The followoing 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.
viIn64(vi, space, offset, val64);	Reads 64 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.
viOut64(vi, space, offset, val64);	Writes 64 bits of data to the specified offset.
viMoveIn8(vi, space, offset, length, buf8);	Moves an 8-bit block of data from the specified offset to local memory.
viMoveIn16(vi, space, offset, length, buf16);	Moves a 16-bit block of data from the specified offset to local memory.
viMoveIn32(vi, space, offset, length, buf32);	Moves a 32-bit block of data from the specified offset to local memory.
viMoveIn64(vi, space, offset, length, buf64);	Moves a 64-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.
viMoveOut64(vi, space, offset, length, buf64);	Moves a 64-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 VI_A64_SPACE - Maps in VXI/MXI A64 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 code sample 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-, 32-, or 64-bit block of data from the specified offset to local memory, and the viMoveOut functions moves an 8-, 16-, 32-, or 64-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-, 32-, or 64-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: Sample Programs

Two sample 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; the second program accesses the backplane with the GPIB-VXI interface. These two programs are identical except for the string passed to viOpen.

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

The <u>vxih1.c sample program</u> uses high-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

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

The <u>gpibvxih.c sample program</u> 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.

/*vxihl.c

This example program uses the high level 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 <stdlib.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);
}
```

/*gpibvxih.c

This example program uses the high level 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 <stdlib.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:

Map memory space using viMapAddress.

Read and write to the register's contents using viPeek32 and viPoke32.

Unmap the memory space using viUnmapAddress.

Low-Level Memory Functions

You can program the registers using low-level functions for 8-, 16-, 32-, or 64bit transfers. The following table summarizes the low-level memory functions.

Description

Function

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.
viPeek32(vi, addr, val32);	Reads 32 bits of data from address specified.
viPeek64(vi, addr, val64);	Reads 64 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.
viPoke64(vi, addr, val64);	Writes 64 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 VI_A64_SPACE - Maps in VXI/MXI A64 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 sample viMapAddress function calls follow.

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

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

Reading and Writing to Device Registers

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

Sample: Using viPeek16

A code sample 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: Code Samples

Two sample 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.

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

The <u>vxill.c sample program</u> uses low-level memory functions and the VXI interface to read the ID and Device Type registers of a device at VXI0::24.

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

The <u>gpibvxil.c sample program</u> 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. /*vxill.c

This example program uses the low level 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 <stdlib.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 */
/* in order 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);
}
```

/*gpibvxil.c

This example program uses the low level 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 <stdlib.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 */
/* in order 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, A32, and A64 address spaces. The best method to use depends on the VISA program characteristics.

- Low-level viPeek/viPoke
 - viMapAddress
 - viUnmapAddress
 - viPeek8, viPeek16, viPeek32, viPeek64
 - viPoke8, viPoke16, viPoke32, viPoke64
- High-level viln/viOut
 - viln8, viln16, viln32, viln64
 - viOut8, viOut16, viOut32, viOut64
- High-level viMoveIn/viMoveOut
 - viMoveIn8, viMoveIn16, viMoveIn32, viMoveIn64
 - viMoveOut8, viMoveOut16, viMoveOut32, viMoveOut64

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 pointers may be directly de-referenced in some cases for the lowest possible overhead.

The disadvantages of low-level viPeek/viPoke are:

- A viMapAddress call is required to set up mapping before viPeek/viPoke can be used.
- viPeek/viPoke calls do not return status codes.
- Only one active viMapAddress is allowed per vi session.
- There may be a limit to the number of simultaneous active viMapAddress calls per process or system.

Using High-Level viIn/viOut

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

The advantages of high-level viIn/viOut are:

- It is the simplest method to implement.
- There is no limit on the number of active maps.
- A16, A24, A32, and A64 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/viMoveOut 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:

- They are simple to use.
- There is no limit on number of active maps.
- A16, A24, A32, and A64 memory access can be mixed in a single vi session.
- They provide the best performance when transferring large blocks of data.
- They support both block and FIFO mode.

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

Sample: Using VXI Memory I/O

The <u>memio.c sample program</u> demonstrates how to use the 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 <stdlib.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 the default resource manager and a session to our instrument */
viOpenDefaultRM (&defaultRM);
viOpen (defaultRM, VXI_INST, VI_NULL,VI_NULL, &vi);

= 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 it is supported */
viGetAttribute( vi, VI ATTR WIN ACCESS, &accessMode );
if ( accessMode == VI_DEREF_ADDR ) {
```

```
/* assign the pointer to a variable of the 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);
```

/*

*/

```
/* 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. Note that 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), then viMoveIn does a block read. If the value of VI_ATTR_SRC_INCREMENT is 0 then viMoveIn does a FIFO read.

If the value of VI_ATTR_DEST_INCREMENT is 1 (the default), then viMoveOut does a block write. If the value of VI_ATTR_DEST_INCREMENT is 0 then viMoveOut does a FIFO write.

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

/*

First set the source increment to 0 so we will repetatively 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, A32, and A64 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 operation <u>viSetAttribute</u>.

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.

Sample: MEMACC Resource Program

The <u>peek16.c sample program</u> 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.

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, GPIB-VXI, and PXI-Specific MEMACC Attributes

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

Attribute	Description
VI_ATTR_SRC_INCREMENT	Used in viMoveInxx operation to specify how much the source offset is to be incremented after every transfer. The default value is 1 and the viMoveInxx operation moves from consecutive elements.
	If this attribute is set to 0, the viMoveInxx operation will always read from the same element, essentially treating the source as a FIFO register.
VI_ATTR_DEST_INCREMENT	Used in viMoveOutxx operation to specify how much the destination offset is to be incremented after every transfer. The default value is 1 and the viMoveOutxx operation moves into consecutive elements.
	If this attribute is set to 0, the viMoveOutxx operation will always write to the same element, essentially treating the destination as a FIFO register.
VI ATTR WIN ACCESS	Specifies modes in which the current window may be addressed: not currently mapped, through the viPeekxx or viPokexx operations only, or through

	operations and/or by directly de-referencing the address parameter as a pointer.
VI_ATTR_WIN_BASE_ADDR_32	Base address of the interface bus to which this window is mapped.
VI_ATTR_WIN_BASE_ADDR_64	Base address of the interface bus to which this window is mapped.
VI_ATTR_WIN_SIZE_32	Size of the region mapped to this window.
VI_ATTR_WIN_SIZE_64	Size of the region mapped to this window.

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

GPIB-VXI-Specific MEMACC Attributes

The following read-only attributes provide specific address information about GPIB hardware.

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.

```
#include <stdio.h> // for printf()
#include <stdlib.h> // for exit()
#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( drm, status, "viOpen", EXIT );
```

```
//
 // Calculate the A16 offset of the VXI registers for the device
 // at VXI logical address 8
 //
 offset = 0xc000 + 64 * 8;
 11
 // Open a map to all of A16 space
 //
 status = viMapAddress( vi, VI_A16_SPACE, 0, 0x10000, VI_FALSE, 0,
(ViPAddr)(&addr));
 checkError( vi, status, "viMapAddress", EXIT );
 //
 // Offset the address pointer returned from viMapAddress for use with
viPeek16
 //
 addr16 = (ViUInt16 *)(addr + offset );
 //
 // Peek the contents of the card's ID register (Offset 0 from card's base address)
 // Note that viPeek does not return a status code.
 //
 viPeek16(vi, addr16, &peekData16);
 //
 // Now use viIn16 and read the contents of the same register
 //
 status = viIn16( vi, VI_A16_SPACE, (ViBusAddress)offset, &inData16 );
 checkError(vi, status, "viIn16", NO EXIT);
 //
 // Print the results
 //
 printf( "inData16 : 0x%04hx\n", inData16 );
 printf( "peekData16: 0x%04hx\n", peekData16 );
 viClose( vi );
 viClose( drm );
}
```

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 VXIspecific 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:

Settings for the VI_ATTR_WIN_ACCESS Attribute

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

Sample: Determining Window Mapping

ViAddr address; Vi UInt16 access; ViUInt16 value;

. . viMapAddress(vi, VI_A16_SPACE, 0x00, 0x04, VI_FALSE,VI_NULL, &address); viGetAttribute(vi, VI_ATTR_WIN_ACCESS, &access);

```
If(access==VI_USE_OPERS) {
  viPeek16(vi, (ViAddr)(((ViUInt16 *)address) + 4/sizeof(ViUInt16)), &value)
}else if (access==VI_DEREF_ADDR){
  value=*((ViUInt16 *)address+4/sizeof(ViUInt16));
}else if (access==VI_NMAPPED){
  return error;
}
.
```

Setting the VXI Trigger Line

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

viSetAttribute(vi, VI_ATTR_TRIG_ID, VI_TRIG_TTL0);

The above function sets the VXI trigger line to TTL trigger line 0 (VI_TRIG_TTL0). The following table shows valid VXI trigger lines.

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

VXI Trigger Line VI_ATTR_TRIG_ID Value

*Panel In is an Agilent extension of the VISA specification.

Once you set a VXI trigger line, you can set up an event handler to be called when the trigger line fires. See <u>Using Events and Handlers</u> 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.

Programming PXI Devices

Agilent VISA supports programming PXI (PCI eXtensions for Instrumentation), PXIe (PCI eXpress eXtensions for Instrumentation), and PCIe (PCI EXPRESS) devices installed in a PC or PXI chassis. These topics describe how to program these devices:

- <u>PXI Overview</u>
- Using High-Level Memory Functions
- Using Low-Level Memory Functions
- <u>Using PXI-Specific Attributes</u>
- Using PXI MEMACC

See <u>Programming with VISA</u>, for general information on VISA programming.

Note: Programming is identical for PXI, PXIe and PCIe device types. The term PXI is used in this document to represent all three device types

The Agilent Modular Driver Wizard

The Agilent Modular Driver Wizard generates device-specific .*ini*, <u>.inf</u>, and *.sys* files for a PXIe or AXIe device.These files enable the Agilent IO Libraries Suite to communicate with the device. You can launch the wizard from <*drive*>:\Program Files\Agilent\IO Libraries Suite\bin\Modular\ModularDriverWizard.exe.

Important This wizard is intended for use by modular hardware

development engineers. It requires knowledge of hardware interrupts and the hardware memory map. If you are an end user (not a module designer) and need a driver for a module, you should refer to the installation software that came with your modular device instead of using the wizard.If, however, you need to regenerate installation files for the device, you can import the device's vendor-supplied *.ini* file in the first step of the wizard. After importing, the wizard can generate and install the files for that device.

See Also

PXI Overview

PXI is a rugged, high-performance modular instrumentation platform designed for industrial measurement and automation applications. PXI allows you to use modules from multiple vendors and easily integrate them into a PXI system.

A PXI device is identified by the PCI bus number where it is located, the assigned PCI device number, and the device's function number. For single-function devices, the function number is 0 and is optional. For multi-function devices, the function number is device specific and ranges from 1 to 7.

The address string for a PXI device is defined as shown below (optional parameters are shown in square brackets []):

"PXI[bus]::device[::function][::INSTR]"

Some examples of PXI device addresses are:

PXI0::3-18::INSTR	PXI device 18 on bus 3
PXI0::3-18.2::INSTR	Function 2 on PXI device 18 on bus 3
PXI0::21::INSTR	PXI device 21 on bus 0
PXI0::CHASSIS1::SLOT4:	:INSTR PXI device in slot 4 of chassis 1
PXI0::MEMACC	Access to system controller memory available

to

devices in the PXI system.

Finding PXI Devices

You can use viFindRsrc() to search for specific resources, such as PXI resources. The <u>VisaFindRsrc() function in the PXIVisaSample program</u> demonstrates how to find all VISA PXI resources.

Each PXI device has a vendor code and model code. You can create a more refined search by adding a vendor and/or model code to the search using the VI_ATTR_MANF_ID attribute . For example, search for an Agilent E2929A PCI/PCI-X Bus Analyzer using Agilent's PCI vendor code (0x15BC) and the model code (0x2929) as follows:

"PXI?*INSTR{VI_ATTR_MANF_ID==0x15BC && VI_ATTR_MODEL_CODE==0x2929}"

PXI Device Types

PXI devices are register based. Programming a PXI device directly from VISA requires a knowledge of the register map of the device and must be done using the VISA memory functions. A PXI 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 and low-level memory functions.
The Agilent Modular Driver Wizard

The Agilent Modular Driver Wizard generates device-specific .*ini*, <u>.inf</u>, and *.sys* files for a PXIe or AXIe device.These files enable the Agilent IO Libraries Suite to communicate with the device. You can launch the wizard from <*drive*>:\Program Files\Agilent\IO Libraries Suite\bin\Modular\ModularDriverWizard.exe.

Important This wizard is intended for use by modular hardware

development engineers. It requires knowledge of hardware interrupts and the hardware memory map. If you are an end user (not a module designer) and need a driver for a module, you should refer to the installation software that came with your modular device instead of using the wizard.If, however, you need to regenerate installation files for the device, you can import the device's vendor-supplied *.ini* file in the first step of the wizard. After importing, the wizard can generate and install the files for that device.

See Also

Using High-Level Memory Functions

High-level memory functions allow you to access memory 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.

High-level memory functions are easier to use; however, the trade-off is speed. 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 <u>Using Low-Level</u> <u>Memory Functions</u>.

Programming the Registers

High-level memory functions include the viIn and viOut functions for transferring 8-, 16-, 32-, or 64-bit values, as well as the viMoveIn and viMoveOut functions for transferring 8-, 16-, 32-, or 64-bit blocks of data into or out of local memory.

High-Level Memory Functions

The table below 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.
viIn64(vi, space, offset, val64);	Reads 64 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.
viOut64(vi, space, offset, val64);	Writes 64 bits of data to the specified offset.
viMoveIn8(vi, space, offset, length, buf8);	Moves an 8-bit block of data from the specified offset to local memory.
viMoveIn16(vi, space, offset, length, buf16);	Moves a 16-bit block of data from the specified offset to local memory.
viMoveIn32(vi, space, offset, length, buf32);	Moves a 32-bit block of data from the specified offset to local memory.
viMoveIn64(vi, space, offset, length, buf32);	Moves a 64-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.
viMoveOut64(vi, space, offset, length, buf32);	Moves a 64-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. Valid space values for PXI devices are:

- VI_PXI_CFG_SPACE Address the PCI configuration space.
- VI_PXI_BAR0_SPACE VI_PXI_BAR5_SPACE Address the specified Base Address Register PCI memory or I/O space.
- VI_PXI_ALLOC_SPACE Access physical locally allocated memory.

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.

Using viMoveIn and viMoveOut

You can 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-, 32-, or 64-bit block of data from the specified offset to local memory, and the viMoveOut functions moves an 8-, 16-, 32-, or 64-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-, 32-, or 64-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 Access Example

The <u>VisaHighLevelMemoryAccess function in the PXIVisaSample program</u> demonstrates how to use viIn32 and viMoveIn32 high-level memory functions.

// PXIVisaSample

// An example of how to use Agilent VISA to get/set attributes

// and access memory on a PXI device.

//

// Error checking should always be done when making VISA calls.

// To simplify this example and make it easier to read,

// most error checking has been eliminated.

#define WIN32_LEAN_AND_MEAN // Exclude rarely-used stuff from Windows headers #include <stdio.h> #include <windows.h> #include "visa.h"

```
// Open a session to the VISA default resource manager
status = viOpenDefaultRM(&drm);
```

```
// Find all PXI resources
char *searchExpression = "PXI?*";
ViSession viFindSession;
ViUInt32 findCount = 0;
ViChar findName[1024];
status = viFindRsrc(drm, searchExpression, &viFindSession, &findCount,
findName);
if (status >= VI_SUCCESS)
{
    printf("reiFindDere of "0", of found 0", d derive(o)", n", searchExpression
```

printf("viFindRsrc of '%s' found %d device(s)\n", searchExpression,

```
findCount);
   for (ViUInt32 index = 1; index <= findCount; index++)</pre>
   {
     printf(" %2d: %s\n", index, findName);
    if (index < findCount)</pre>
     {
      status = viFindNext(viFindSession, findName);
     }
   }
   status = viClose(viFindSession);
 }
 status = viClose(drm);
 printf("\n");
}
void VisaLowLevelMemoryAccess(char *visaName)
{
 ViStatus status;
 ViSession drm;
 printf("VISA Low Level Memory Access Example\n");
 // Open a session to the VISA default resource manager
 status = viOpenDefaultRM(&drm);
 // Open a session to the PXI device
 ViSession vi;
 status = viOpen(drm, visaName, VI NULL, VI NULL, &vi);
 printf("Opening '%s' returned status=0x%x\n", visaName, status);
 // Get the manufacturer and model name
 ViChar manfName[1024];
 ViChar modelName[1024];
 status = viGetAttribute(vi, VI_ATTR_MANF_NAME, manfName);
 status = viGetAttribute(vi, VI_ATTR_MODEL_NAME, modelName);
 printf(" Manufacturer Name: '%s'\n", manfName);
```

```
printf(" Model Name: '%s'\n", modelName);
```

```
// Get the size of the BAR0 memory
```

else if (winAccess == VI NMAPPED)

```
ViAttr mapAttr = VI_ATTR_PXI_MEM_SIZE_BAR0;
ViUInt16 mapSpace = VI_PXI_BAR0_SPACE;
ViBusSize mapSize = 0;
status = viGetAttribute(vi, mapAttr, &mapSize);
```

```
// Map BAR0 memory - Note: only one map at a time is allowed per VISA
session
```

```
ViBusAddress mapOffset = 0;
 ViAddr
            mapAddress = 0;
 status = viMapAddress(vi, mapSpace, mapOffset, mapSize, VI_NULL,
VI NULL, &mapAddress);
 if (sizeof(void*)==8)
 {
   // ViBusSize and viBusAddress are 64-bit values in 64-bit applications
   printf(" Size of BAR0 memory = %I64d bytes\n", mapSize);
   printf(" BAR0 Map Address = 0x%016I64x\n", mapAddress);
 }
 else
 {
   // ViBusSize and viBusAddress are 32-bit values in 32-bit applications
   printf(" Size of BAR0 memory = %d bytes\n", mapSize);
   printf(" BAR0 Map Address = 0x%08x\n", mapAddress);
 }
 // Find out if BAR0 memory can be dereferenced directly
 ViUInt16 winAccess;
 status = viGetAttribute(vi, VI_ATTR_WIN_ACCESS, &winAccess);
 if (winAccess == VI_DEREF_ADDR)
 ł
   printf(" BAR0 memory can be directly dereferenced\n");
 }
 else if (winAccess == VI USE OPERS)
 {
   printf(" BAR0 memory cannot be directly dereferenced.\n");
```

```
{
   printf(" BAR0 memory is not currently mapped.\n");
 if (winAccess == VI_DEREF_ADDR)
 {
   printf(" Dereferencing BAR0 memory\n");
   // Here are some examples of direct memory dereferencing:
   ViUInt32 memValue0 = *((ViUInt32 *)mapAddress);
   ViUInt32 memValue1 = *((ViUInt32 *)mapAddress+1);
             memValue0 = 0x\%08x, memValue1 = 0x\%08x\n'', memValue0,
   printf("
memValue1);
   ViChar memValueArray[16];
   memcpy(memValueArray, mapAddress, sizeof(memValueArray));
             memValueArray = ");
   printf("
   for (int index = 0; index < sizeof(memValueArray); index++)</pre>
   {
     printf("%02x", memValueArray[index]);
   }
   printf("\n");
  }
 if ((winAccess == VI_USE_OPERS) || (winAccess == VI_DEREF_ADDR))
  {
   printf(" Using viPeek on BAR0 memory\n");
   // Some examples of using viPeek for memory access:
   ViUInt32 memValue0;
   ViUInt32 memValue1:
   viPeek32(vi, mapAddress, &memValue0);
   viPeek32(vi, ((ViUInt32 *)mapAddress)+1, &memValue1);
             memValue0 = 0x\%08x, memValue1 = 0x\%08x\n'', memValue0,
   printf("
memValue1);
  }
 status = viUnmapAddress(vi);
 status = viClose(vi);
```

```
status = viClose(drm);
printf("\n");
}
```

```
void VisaHighLevelMemoryAccess(char *visaName)
```

```
{
```

```
// Open a session to the PXI device
ViSession vi;
status = viOpen(drm, visaName, VI_NULL, VI_NULL, &vi);
printf("Opening '%s' returned status=0x%x\n", visaName, status);
```

```
// Get the manufacturer and model name
```

```
ViChar manfName[1024];
ViChar modelName[1024];
status = viGetAttribute(vi, VI_ATTR_MANF_NAME, manfName);
status = viGetAttribute(vi, VI_ATTR_MODEL_NAME, modelName);
printf(" Manufacturer Name: '%s'\n", manfName);
printf(" Model Name: '%s'\n", modelName);
```

```
ViUInt16 mapSpace = VI_PXI_BAR0_SPACE;
ViUInt32 memValue0;
ViUInt32 memValue1;
ViUInt32 memValueArray[4];
```

```
// viIn example
status = viIn32(vi, mapSpace, 0, &memValue0);
status = viIn32(vi, mapSpace, sizeof(ViUInt32) * 1, &memValue1);
printf(" viIn32 from BAR0 memory\n");
printf(" memValue0 = 0x%08x, memValue1 = 0x%08x\n", memValue0,
```

memValue1);

// Show the default DMA value and set it to VI_TRUE so viMoveIn32 will use DMA if

```
// the device supports it.
 ViBoolean allowDma;
 status = viGetAttribute(vi, VI ATTR DMA ALLOW EN, &allowDma);
 printf(" Current value of VI ATTR DMA ALLOW EN is %s\n",
     (allowDma == VI_TRUE)? "VI_TRUE" : "VI_FALSE");
 status = viSetAttribute(vi, VI ATTR DMA ALLOW EN, VI TRUE);
 status = viGetAttribute(vi, VI ATTR DMA ALLOW EN, &allowDma);
 printf(" After setting it VI ATTR DMA ALLOW EN is %s\n",
     (allowDma == VI_TRUE)? "VI_TRUE" : "VI_FALSE");
 // viMoveIn example
 status = viMoveIn32(vi, mapSpace, 0,
            sizeof(memValueArray)/sizeof(memValueArray[0]),
            memValueArray);
 printf(" viMoveIn32 from BAR0 memory\n");
 printf("
           memValueArray = ");
 for (int index = 0; index < sizeof(memValueArray)/sizeof(memValueArray[0]);</pre>
index++)
 {
   printf("0x%08x ", memValueArray[index]);
 }
 printf("\n");
 status = viClose(vi);
 status = viClose(drm);
 printf("\n");
}
void main()
{
 VisaFindRsrc();
```

char *visaName = "pxi0::1-1.0::instr";

VisaLowLevelMemoryAccess(visaName); VisaHighLevelMemoryAccess(visaName);

}

Using Low-Level Memory Functions

Low-level memory functions allow direct access to memory, 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 improve. 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.

Low-Level Memory Functions

You can program the registers using low-level functions for 8-, 16-, 32-, or 64bit transfers. The table below summarizes the low-level memory functions.

Description

Function

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.
viPeek32(vi, addr, val32);	Reads 32 bits of data from address specified.
viPeek64(vi, addr, val64);	Reads 64 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.
viPoke64(vi, addr, val64);	Writes 64 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 specifies the memory location to map; the following are valid mapSpace choices:

VI_PXI_CFG_SPACE - Address the PCI configuration space.

VI_PXI_BAR0_SPACE - VI_PXI_BAR5_SPACE - Address the specified Base Address Register PCI memory or I/O space.

VI_PXI_ALLOC_SPACE - Access physical locally allocated memory.

A pointer to the address space where the memory was mapped is returned in the address parameter. If the device specified does not have memory in the specified address space, an error is returned. A sample viMapAddress function call follows.

/* Maps to VI_PXI_BAR0_SPACE address space */ viMapAddress(vi, VI_PXI_BAR0_SPACE 0x000, 0x100, VI_FALSE, VI_NULL,&address);

Note: When calling the viMapAddress function on a PXI session, the maximum value for the mapSize parameter is 1048576 (0x100000), even if the map space being referenced is larger than this value. If you need access to a larger memory block than this, there are two ways to work around this limit:

1.Use viMoveIn, viMoveOut, or viMove. These functions are not limited to the 0x100000 byte limit.

2.Use viMapAddress / viUnmapAddress multiple times to map individual blocks of memory that are less than 0x100000 bytes and access each block in sequence.

Determining Window Mapping

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 pointer if the address is dereferenced. To determine how to access the window, read the VI_ATTR_WIN_ACCESS attribute.

VI_ATTR_WIN_ACCESS Settings

Description

Setting

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

C	•
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_ADDF	Specifies that the window is mapped and has a de-referenced address. In this case you can use the a 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.

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 instrument's user manual for a description of the registers and register locations. You can then use this information and the VISA low-level functions to access the device registers.

Low-Level Memory Access Example

The <u>VisaLowLevelMemoryAccess function in the PXIVisaSample program</u> shows some direct memory dereferencing examples, and some viPeek examples.

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.

Using PXI-Specific Attributes

PXI-specific attributes are useful to determine device settings such as the PCI bus, device, and function numbers, the size of BARn memory, and so on. For example, to determine the size of BAR0 memory:

// Get the size of the BAR0 memory

ViAttr mapAttr = VI_ATTR_PXI_MEM_SIZE_BAR0; ViUInt16 mapSpace = VI_PXI_BAR0_SPACE; ViBusSize mapSize = 0; status = viGetAttribute(vi, mapAttr, &mapSize);

PXI-Specific Attributes

VI_ATTR_PXI_BUS_NUM - PCI bus number of this device.

VI_ATTR_PXI_DEV_NUM - PCI device number of this device.

VI_ATTR_PXI_FUNC_NUM - PCI function number of the device. All devices have a function 0. Multifunction devices also support other function numbers.

VI_ATTR_PXI_SLOTPATH - Slot path of this device. A PXI slot path is a sequence of values representing the PCI device number and function number of a PCI module and each parent PCI bridge that routes the module to the host PCI bridge. The string format of the attribute value is device1[.function1] [,device2[.function2]][,...].

VI_ATTR_PXI_SLOT_LBUS_LEFT - Slot number or special feature connected to the local bus left lines of this device.

VI_ATTR_PXI_SLOT_LBUS_RIGHT - Slot number or special feature connected to the local bus right lines of this device.

VI_ATTR_PXI_TRIG_BUS - Number of the trigger bus connected to this device in the chassis.

VI_ATTR_PXI_STAR_TRIG_BUS - Number of the star trigger bus connected to this device in the chassis.

VI_ATTR_PXI_STAR_TRIG_LINE - PXI_STAR line connected to this device.

VI_ATTR_PXI_MEM_TYPE_BARn - Memory type (memory mapped or I/O mapped) used by the device in the specified BAR.

VI_ATTR_PXI_MEM_BASE_BARn - Memory base address assigned to the specified BAR for this device.

VI_ATTR_PXI_MEM_SIZE_BARn - Size of the memory assigned to the specified BAR for this device.

VI_ATTR_PXI_CHASSIS - Chassis number in which this device is located.

VI_ATTR_PXI_IS_EXPRESS - Specifies whether this device is PXI Express.

VI_ATTR_PXI_SLOT_LWIDTH - Specifies the link width used by the slot in which this device is located.

VI_ATTR_PXI_MAX_LWIDTH - Specifies the maximum link width that this device can use.

VI_ATTR_PXI_ACTUAL_LWIDTH - Specifies the negotiated link width that this device is using.

VI_ATTR_PXI_DSTAR_BUS - Number of the DSTAR bus connected to this device in the chassis.

VI_ATTR_PXI_DSTAR_SET - Specifies the set of PXI_DSTAR lines connected to this device.

Using PXI MEMACC

The PXI MEMACC resource provides the attributes, events, and operations necessary to allow controller software and PXI devices to access PXI system controller memory. The address string for PXI MEMACC is:

PXI0::MEMACC Access to system controller memory available to devices in the PXI system.

For more information, refer to <u>Using the Memory Access Resource</u>.

Programming via LAN

This section describes how to program instruments via a LAN (Local Area Network). Topics are:

- LAN and Remote Interfaces Overview
- Using the TCPIP Interface Type for LAN Access
- Using a Remote Interface for LAN Access

LAN and Remote Interfaces Overview

This topic contains 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 with instruments over the LAN, you must first configure a LAN interface or a remote GPIB, USB, or serial interface, using the Agilent Connection Expert.

Direct LAN Connection versus Remote IO Server/Client Connection

Some instruments support direct connection to the LAN. These instruments include an RJ-45 or other standard LAN connector and software support for operating as an independent device on the network. Some of these instruments are Web-enabled, meaning that they host a Web page which you can access over the LAN.

With the Agilent IO Libraries Suite, you can connect to instruments across the LAN even if they do not have direct LAN capability, if they are connected to gateways (such as the Agilent E5810A) or to another PC running the Remote IO Server software.

Refer to the IO Libraries Suite and the Connectivity Guide for information on connecting and configuring different types of LAN instrument connections.

Remote IO Server/Client Architecture

The Remote IO Server and Client software provided with Agilent IO Libraries Suite allows instrumentation to be controlled 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 IO Libraries Suite software uses the client/server model of computing. Client/server computing refers to a model in which an application (the client) does not perform all necessary tasks of the application itself. Instead, the client makes requests of another computing device (the remote I/O server) for certain services.

As shown in the following figure, a remote I/O client (a Windows PC) makes VISA requests over the network to a remote I/O server (such as a Windows PC, an E5810A LAN/GPIB Gateway, or a Series 700 HP-UX workstation).

Gateway Operation

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

Windows PCs



Using the TCPIP Interface Type for LAN Access

VISA provides the TCPIP interface type to communicate with LAN-connected devices. These can be devices connected directly to the LAN, or they can be connected to the LAN through a LAN gateway, such as the Agilent E5810A LAN/GPIB gateway, or through Remote IO Server software running on a remote computer with instruments connected to it.

The TCPIP interface type supports these protocols:

- VXI-11 (TCP/IP Instrument Protocol) is a networking protocol, developed by the VXIbus Consortium, which permits interoperability of LAN software from different vendors. This protocol supports SICL and VISA operations to a LAN-based VXI-11 instrument, SICL operations over the LAN to GPIB or VXI interfaces (message-based devices only), and VISA operations over the LAN to GPIB interfaces. The VXI-11 protocol does not support serial or USB interfaces.
- High-Speed LAN Instrument Protocol (HiSLIP) is a protocol for TCPbased instrument control that provides high-speed performance while maintaining the instrument-like capabilities of conventional test and measurement protocols.
- SICL-LAN is an older networking protocol developed by Agilent Technologies. The SICL-LAN protocol supports SICL operations over the LAN to GPIB, VXI (message-based devices only), USB, and RS-232 interfaces.
- Raw socket connection is a protocol for instruments that do not support a higher-level protocol. The SOCKET resource exposes the capability of a raw network socket connection over TCP/IP. This usually means Ethernet but the protocol is not restricted to that physical interface. Services are provided to send and receive blocks of data.

Like HiSLIP, sockets is a "fire and forget" sender protocol. This means that when a command is sent over sockets (or HiSLIP), the protocol writes it to the connection, but does not verify that the command has been received or processed by the instrument before returning from the send. This can sometimes cause command execution problems when making multiple connections to an instrument, see <u>Multiple Connections</u> for ways to ensure proper command execution.

The protocol(s) you will use depends upon the devices you are using and the protocol(s) that they support. Many instruments support multiple protocols.

Using the SICL-LAN or VXI-11 Protocol

For VXI-11 or SICL-LAN, the format of a TCPIP VISA resource string is:

TCPIP[board]::hostname[::LAN device name][::INSTR]

where:

- board = board number (default is 0)
- hostname = the hostname or IPv4 IP address of the LAN device or server. Note: IPv6 addressing cannot be used with the VXI-11 or SICL-LAN protocols. IPv6 is supported for HiSLIP and Raw Sockets.
- LAN device name = the remote device name (case sensitive with default name of inst0)

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

The SICL-LAN protocol allows any valid SICL name for the LAN device name. A valid SICL name must be a unique string of alphanumeric characters, starting with a letter.

The following table shows some examples of TCPIP resource strings.

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

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

connected to a LAN server at hostname myMachine.

This device must use SICL-LAN protocol since USB devices are not supported by the VXI-11 protocol.

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

Addressing a Session Using the TCPIP Interface Type

This sample shows one way to open a device session with a GPIB device at primary address 23 on a remote PC that is running a LAN server. The hostname of the remote PC is myMachine. See <u>Programming with VISA</u>, for more information on addressing device sessions.

ViSession defaultRM, vi;.

viOpenDefaultRM(&defaultRM);

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

•

•

viClose(vi);

viClose(defaultRM);

Using the HiSLIP Protocol

HiSLIP works at nearly the speed of raw sockets, allowing fast query/response transactions and fast bulk data transfers while still supporting the instrument-like capabilities of conventional test and measurement protocols. The HiSLIP protocol is similar to VXI-11; however, there are differences, see <u>Comparing HiSLIP and VXI-11 Systems</u> for details.

HiSLIP Features

- In addition to high-performance with instrument-like capabilities, HiSLIP has the following features.
- Reliable end of message (EOM) signaling, regardless of data sent.
- Asynchronous service request (SRQ) signaling (viInstallHandler, viEnableEvent support VI_EVENT_SERVICE_REQ).
- Status byte (viReadStb) readable regardless of the data/command state of the instrument, including when the instrument is hanging on a command).
- Device Clear (viClear) clears the connection of data and commands, including when the instrument is hanging on a command.
- Supports remote/local status and control (viGpibControlREN, VI_ATTR_GPIB_ADDR_STATE, and VI_ATTR_GPIB_REN_STATE).
- Enhanced locking:
 - Shared and exclusive locks are maintained in the instrument, allowing locks to be honored by all hosts attempting HiSLIP connections to the instrument.
 - Delayed lock error reporting allows more natural lock enforcement. See <u>HiSLIP Locking Behavior</u> for details.
 - Stale responses are detected and suppressed via interrupted error handling.
- Supports both IPv4 (Internet Protocol version 4) and IPv6 (Internet Protocol version 6) connections. IPv4 uses a 32-bit address space; IPv6 uses a 128-bit address space.
- For more information on the HiSLIP protocol, see the HiSLIP specification and the latest VISA specification. Both specifications are available at www.ivifoundation.org.

Identifying HiSLIP Devices

An instrument that supports HiSLIP is identified in the system by its VISA address. The address syntax is

TCPIP[board]::host address[::HiSLIP device name[,HiSLIP port]][::INSTR]

Note: The HiSLIP device name must begin with hislip to establish a HiSLIP session.

For example, a typical HiSLIP VISA address for a TCP/IP device located at IPv6 IP address fe80::1 is:

TCPIP0::[fe80::1]::hislip0::INSTR

HiSLIP VISA Attributes

The HiSLIP protocol introduces three new VISA attributes for HiSLIP devices:

- VI_ATTR_TCPIP_HISLIP_OVERLAP_EN If VI_TRUE, enables overlap mode for HiSLIP. If VI_FALSE, the synchronous mode is used. Refer to <u>Synchronous and Overlap Modes</u> for a description of these modes.
- VI_ATTR_TCPIP_HISLIP_VERSION Returns the ViVersion in use by the current HiSLIP session. For example, HiSLIP version 1.0 returns a ViVersion value of 0x00100000.
- VI_ATTR_TCPIP_HISLIP_MAX_MESSAGE_KB Specifies the largest size HiSLIP message VISA will accept in units of kilobytes; defaults to 1024 (a 1 MB maximum message size).

HiSLIP Locking Behavior

The HiSLIP protocol introduces these new locking behaviors:

- Lock enforcement for HiSLIP devices is done by the HiSLIP device.
 - viLock() pushes shared and exclusive locks requested on HiSLIP connections to the HiSLIP device, with the exception of nested locks.
 - Lock-respecting VISA operations are sent to the device.

- If the HiSLIP device is locked and the current session does not have the lock, the operation is delayed until the lock is released.
- If VISA times out on an operation on a HiSLIP connection, VISA checks whether the HiSLIP device is locked. If a lock delayed the operation too long, it returns a (delayed) VI_ERROR_RSRC_LOCKED return value.
- Device Clear on a HiSLIP connection clears any queued operations delayed by a lock.
- viUnlock() sends an unlock request to the HiSLIP device as the last of any nested locks is unlocked.
- VI_RSRC_LOCK_STATE [VI_ATTR_RSRC_LOCK_STATE] now returns the lock state across all open connections of the same interface and protocol. For HiSLIP, this requires querying the HiSLIP device for the lock state.

Comparing HiSLIP and VXI-11 Systems

HiSLIP and VXI-11 systems are similar in many ways. For example, both systems support sub-instrument addressing. HiSLIP can be used as a functional replacement for VXI-11, but there are some differences:

- **Performance:** HiSLIP is significantly faster than VXI-11. If your program has timing dependencies based on VXI-11 performance, you may need to change it to accommodate the faster HiSLIP.
- Multiple connections: HiSLIP permits many simultaneous active connections, whereas VXI-11 only allows one connection at a time. It is your responsibility to coordinate instrument operations on multiple connections, for example by using locks (see below). Locks guarantee that the operations done while the lock is active are complete before the lock is released. That ensures that all instrument changes made by a HiSLIP connection holding the lock are complete before any other HiSLIP connection can have its commands executed. Even if your application ensures HiSLIP connections do not overlap, you should make sure that all operations requested by a particular HiSLIP connection are complete before closing that connection (for example by using *OPC? at the end of the command sequence on the connection). Otherwise, the next HiSLIP connection might interfere with the completion of the last
HiSLIP connection's instrument operations.

- Fire and forget: Like sockets, HiSLIP is a "fire and forget" sender protocol. This means that when a command is sent over HiSLIP (or sockets), the protocol writes it to the connection, but does not verify that the command has been received or processed by the instrument before returning from the send. By contrast, VXI-11 handshakes each command, so that the send does not return until the command has been received at the VXI-11 driver layer in the instrument.
- Locking:
 - Shared locks: HiSLIP holds both shared and exclusive locks in the instrument. VXI-11 holds exclusive locks in the instrument, but shared locks in the VISA layer. Because of this, HiSLIP can coordinate shared locks across multiple test controllers, whereas VXI-11 cannot.
 - Lock errors: HiSLIP utilizes "lazy" lock errors: If an instrument is locked when a command is sent via HiSLIP, the sent command is buffered in the instrument and the protocol waits briefly, to allow the lock to be released. If the lock is released in a timely manner, there will be no error. If not, a subsequent viRead may time out and return a lock error. In VXI-11, lock errors are returned immediately.
 - **HiSLIP and VXI-11 locks are independent:** A HiSLIP lock will not lock out operations on a VXI-11 connection, nor will a VXI-11 lock affect operations on a HiSLIP connection.
 - IPv6 support: HiSLIP supports both IPv4 (Internet Protocol version 4) and IPv6 (Internet Protocol version 6) connections.
 IPv4 uses a 32-bit address space; IPv6 uses a 128-bit address space. VXI-11 does not support IPv6.
- Interrupted errors: HiSLIP detects and corrects interrupted errors by default, but can also be operated in an overlapped mode where interrupted errors are ignored but responses are sent as quickly as possible from the HiSLIP system.
- **Modes:** The HiSLIP synchronous mode is similar to VXI-11, GPIB, and USBTMC behavior. Overlap mode is unique to HiSLIP. Both synchronous and overlapped modes are discussed in the next section.

Synchronous and Overlap Modes

In synchronous mode, HiSLIP detects and corrects for interrupted errors. In overlap mode, HiSLIP ignores interrupted errors and sends responses as quickly as possible. The mode is controlled by the VI_ATTR_TCPIP_HISLIP_OVERLAP_EN attribute.

Synchronous Mode

In synchronous mode, VISA and the instrument work together to discard stale responses. For example, if your program sends two queries, but does just one read, the first response is discarded and the second response is read. In some cases, when an interrupted error is detected by the instrument, an error is sent to the instrument's error log.

Overlap Mode

Overlap mode allows multiple operations to be initiated and conducted independently of the rate the device processes messages and sends responses. In overlap mode, stale responses (interrupted errors) are ignored and the instrument is free to send responses as soon as they occur. The responses are returned in the same order the messages were sent. Your test application is responsible for associating the responses with the queries that generated them (by reading the responses in the order that the queries were sent).

Creating a HiSLIP Program

Since HiSLIP programming is similar to VXI-ll programming, you can start with any sample program for an instrument that uses an IO interface and change the address string. For example, change the address string in the viOpen() call to:

```
err = viOpen(rmSession, "TCPIP0::myInstHostname::hislip0::INSTR",
VI_NO_LOCK, 4000, &viSession);
```

Using a Remote Interface for LAN Access

Agilent VISA provides three types of VISA LAN Client interfaces, implemented in Agilent IO Libraries Suite as remote interfaces:

- Remote serial interface (ASRL VISA LAN Client)
- Remote GPIB interface (GPIB VISA LAN Client)
- Remote USB interface (USB VISA LAN Client)

Remote interfaces are configured using Connection Expert; they provide virtual GPIB, serial, or USB interfaces. They make it possible to remotely access a LAN-connected device as if it were connected to a local interface. If, for example, the GPIB2 interface is configured as a remote GPIB interface, a program controlling the devices GPIB2::5::INSTR and GPIB2::7::INSTR would not be aware of the fact that these devices are actually connected via LAN and not to a GPIB interface connected to the local machine.

Note: See the Agilent IO Libraries Suite Help for specific information on configuring remote interfaces.

Remote Serial Interface (ASRL VISA LAN Client)

A remote serial interface can use only the SICL-LAN protocol. A remote serial interface can be configured to use the serial port on the Agilent E5810A LAN/GPIB gateway or the serial ports on a PC running the Remote IO Server software.

Remote GPIB Interface (GPIB VISA LAN Client)

A remote GPIB interface can use both the VXI-11 and SICL-LAN protocols. Typical uses for remote GPIB interfaces are with LAN/GPIB gateways (for example, Agilent E5810A), PCs with GPIB interfaces that are running a LAN server, and VXI-11.2 LAN-based instruments.

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

so they cannot be accessed with a remote GPIB interface.

Remote USB Interface (USB VISA LAN Client)

A remote USB interface can use only the SICL-LAN protocol. It can communicate with USB devices attached to a remote PC running the Remote IO Server software.

Note that if you have defined a VISA alias for a USB device on the remote I/O server, you must either define the same (or another) alias for the remote USB device on the client PC, or use the full USB resource string. Alias definitions are not shared between the remote I/O server and the client.

Addressing a Session Using a Remote Interface

In general, the rules to address a remote session are the same as to address a local session. The only difference for a remote session is that you use the VISA interface ID (provided during I/O configuration via Connection Expert) that relates to the remote interface.

The following sample shows one way to open a device session with a GPIB device at primary address 23 on a remote PC that is running Remote IO Server software. A remote GPIB interface has been configured at GPIB2 to communicate with that machine. See <u>Programming with VISA</u>, for more information on addressing device sessions.

ViSession defaultRM, vi;.

viOpenDefaultRM(&defaultRM);

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

.

viClose(vi);

viClose(defaultRM);

Programming via USB

This section provides describes how to program USB instruments that conform to USBTMC (Universal Serial Bus Test and Measurement Class) and/or USBTMC-USB488 (Universal Serial Bus Test and Measurement Class, Subclass USB488 Specification). Topics are:

- USB Interfaces Overview
- <u>Communicating with a USB Instrument Using VISA</u>

USB Interfaces Overview

USBTMC/USBTMC-USB488 instruments are detected and automatically configured by Agilent VISA when they are plugged into the computer. The Agilent IO Libraries Suite Help describes the USB instrument configuration process in more detail.

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

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

You can also create VISA aliases for other (non-USB) instruments, using the Agilent Connection Expert.

Communicating with a USB Instrument Using VISA

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

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

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

Following is a summary of the components of this call.

Value	Description	Data Type
2391	Manufacturer ID	16-bit unsigned integer
1031	Model Code	16-bit unsigned integer
000000123	Serial Number	string value
0	USBTMC Interface Number	8-bit unsigned integer

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

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

Although the case of a VISA alias is preserved, case is ignored when the alias is used in place of the full resource string in a viOpen call. For example, UsbDevice1, usbdevice1 and USBDEVICE1 all refer to the same device.

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

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

This is much simpler than having to use the full resource string for a USB device.

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

List of VISA Functions

This lists 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.

Function Usage

Opening/Closing Sessions

viOpenDefaultRM viOpen viClose Open default RM session Open session Close session

Control

<u>viAssertTrigger</u>	Assert software or hardware trigger
<u>viAssertUtilSignal</u>	Asserts the specified utility bus signal
<u>viGetAttribute</u>	Get attribute
<u>viMemAlloc</u>	Allocate memory from a device's memory region
<u>viMemAllocEx</u>	Allocate memory from a device's memory region
<u>viMemFree</u>	Free memory previously allocated using viMemAlloc() or viMemAllocEx().
<u>viMemFreeEx</u>	Free memory previously allocated using viMemAlloc() or viMemAllocEx().
<u>viSetAttribute</u>	Set attribute
<u>viStatusDesc</u>	Get status code description
<u>viTerminate</u>	Terminate asynchronous operation
<u>viLock</u>	Lock resource
<u>viUnlock</u>	Unlock resource
<u>viMapTrigger</u>	Map trigger source line to destination line
<u>viUnmapTrigger</u>	Map trigger line to another trigger line

Event Handling/Interrupts

Asserts the specified device interrupt or signal
Enable event
Disable event
Discard events
Wait on event
Install handler
Uninstall handler
Event handler prototype

VXI Specific Series

viVxiCommandQuery

Send device a command/query and/o retrieve a response

Searching

<u>viFindRsrc</u>	Find device
<u>viFindNext</u>	Find next device
<u>viParseRsrc</u>	Parse resource string to get interface information
<u>viParseRsrcEx</u>	Parse resource string to get extended interface information

Basic I/O

<u>viRead</u>	Read data from device
<u>viWrite</u>	Write data to device
<u>viReadAsync</u>	Read data asynchronously from device
<u>viWriteAsync</u>	Write data asynchronously to device
<u>viClear</u>	Clear a device
<u>viReadToFile</u>	Read data synchronously and store data in file
<u>viWriteFromFile</u>	Write data from file synchronously
<u>viAssertTrigger</u>	Assert software/hardware trigger
<u>viReadSTB</u>	Read status byte

Formatted I/O

viSetBuf	Set size of buffer
<u>viBufRead</u>	Unformatted read to formatted I/O buffers
<u>viBufWrite</u>	Unformatted write to formatted I/O buffers
viFlush	Flush read and write buffers
<u>viPrintf</u>	Convert, format, and send parameters to a device
<u>viSPrintf</u>	Write data to a buffer
<u>viVSPrintf</u>	Convert, format, and send parameters to a buffer
<u>viScanf</u>	Receive data from device, format and store data
<u>viVScanf</u>	Receive data from device, format and store data
viSScanf	Receive data from buffer, format and store data
<u>viVSScanf</u>	Receive data from buffer, format and store data
<u>viQueryf</u>	Formatted write and read operation
viVQueryf	Formatted write and read operation

Memory I/O

<u>viIn8</u>	Read 8-bit value from memory space
<u>viIn16</u>	Read 16-bit value from memory space
<u>viIn32</u>	Read 32-bit value from memory space
<u>viIn64</u>	Read 64-bit value from memory space
<u>viOut8</u>	Write 8-bit value to memory space
<u>viOut16</u>	Write 16-bit value to memory space
viOut32	Write 32-bit value to memory space
viOut64	Write 64-bit value to memory space
<u>viMove</u>	Move data from source to destination
<u>viMoveEx</u>	Move a block of data
<u>viMoveAsync</u>	Move data from source to destination asynchronously
<u>viMoveAsyncEx</u>	Move a block of data asynchronously.
<u>viMoveIn8</u>	Move 8-bit value from device memory to local memory
<u>viMoveIn16</u>	Move 16-bit value from device memory to local memory
<u>viMoveIn32</u>	Move 32-bit value from device memory to local memory
<u>viMoveIn64</u>	Move 64-bit value from device memory to local memory
<u>viMoveOut8</u>	Move 8-bit value from local memory to device memory
<u>viMoveIn8Ex</u>	Move a block of data from the specified address space and offset to local memory in increments of 8 bits.
<u>viMoveIn16Ex</u>	Move a block of data from the specified address space and offset to local memory in increments of 16 bits.
<u>viMoveIn32Ex</u>	Move a block of data from the specified address space and offset to local memory in increments of 32 bits.
<u>viMoveIn64Ex</u>	Move a block of data from the specified address space and offset to local memory in increments of 64 bits.
<u>viMoveOut16</u>	Move 16-bit value from local memory to device memory
viMoveOut32	Move 32-bit value from local memory to device memory
viMoveOut64	Move 64-bit value from local memory to device memory
<u>viMoveOut8EX</u>	Move a block of data from local memory to the specified address space and offset in increments of 8 bits.
<u>viMoveOut16EX</u>	Move a block of data from local memory to the specified address space and offset in increments of 16 bits.
<u>viMoveOut32EX</u>	Move a block of data from local memory to the specified address space and offset in increments of 32 bits.
<u>viMoveOut64EX</u>	Move a block of data from local memory to the specified address space and offset in increments of 64 bits.

<u>viMapAddress</u>	Map memory space
<u>viMapAddressEx</u>	Map memory space
<u>viUnmapAddress</u>	Unmap memory space
<u>viPeek8</u>	Read 8-bit value from address
<u>viPeek16</u>	Read 16-bit value from address
viPeek32	Read 32-bit value from address
<u>viPeek64</u>	Read 64-bit value from address
<u>viPoke8</u>	Write 8-bit value to address
<u>viPoke16</u>	Write 16-bit value to address
<u>viPoke32</u>	Write 32-bit value to address
<u>viPoke64</u>	Write 64-bit value to address

Interface Specific Services

viGpibControlREN viGpibControlATN viGpibCommand viGpibPassControl viGpibSendIFC Control GPIB REN interface line Control GPIB ATN interface line Write GPIB command bytes on the bus Tell GPIB device to become Controller in Charge (CIC) Pulse Interface Clear (IFC) line

USB Specific Services

viUsbControlIn viUsbControlOut Request arbitrary data from the USB device on the control port Send arbitrary data to the USB device on the control port

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	Dir	Туре	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
statusID	IN	ViUInt32	This is the status value to be presented during an interrupt acknowledge cycle.

Special Values for *mode* **Parameter**

mode	Action Description	
VI_ASSERT_IRQ1 - VI_ASSERT_IRQ7	VXI/VME IRQ line. This uses the standard VXI/VME ROAK (release on acknomechanism rather than the older VME RORA (release on register access) mechan	
VI_ASSERT_SIGNAL	Send the notification via a VXI signal.	
VI_ASSERT_USE_ASSIGNED	Use whatever notification method that has been assigned to the local device.	

Return Values

Type ViStatus	This is the function return status. It returns either a completion code c follows.	
Completion Code	Description	
VI_SUCCESS	Operation completed successfully.	
Error Codes	Description	
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_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same val	
VI_ERROR_NSUP_INTR	The interface cannot generate an interrupt on the requested level or w <i>statusID</i> value.	
VI_ERROR_NSUP_MODE	The specified mode is not supported by this VISA implementation.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource iden been locked for this kind of access.	

See Also

BACKPLANE resource description

viAssertTrigger

Syntax

viAssertTrigger(ViSession vi, ViUInt16 protocol);

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. For a session to a USB instrument, this function sends the TRIGGER message ID on the Bulk-OUT pipe.

For GPIB, ASRL, USB, and VXI software triggers,

VI_TRIG_PROT_DEFAULT is the only valid protocol. For VXI hardware triggers, VI_TRIG_PROT_DEFAULT is equivalent to VI_TRIG_PROT_SYNC.

For a PXI resource, viAssertTrigger is not currently supported. Support is planned for a later release of Agilent VISA.

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

Parameters

Name	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
			Trigger protocol to use during assertion. Valid values are:
			VI_TRIG_PROT_DEFAULT
			VI_TRIG_PROT_ON - asserts the trigger
protocol	IN	ViUInt16	VI_TRIG_PROT_OFF - deasserts the trigger
			VI_TRIG_PROT_SYNC - pulses the trigger (assert followed by deassert)
			For GPIB, ASRL, USB, and VXI software triggers, VI_TRIG_PROT_DEFAULT is protocol. For VXI hardware triggers, VI_TRIG_PROT_DEFAULT is equivalent to VI_TRIG_PROT_SYNC.

Return Values

Type ViStatus	This is the function return status. It returns either a completion cor an error code as follows. Description	
Completion Code		
VI_SUCCESS	The specified trigger was successfully asserted to the device.	
Error Codes	Description	
VI_ERROR_BERR	Bus error occurred during transfer.	
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.	
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.	
VI_ERROR_INV_PROT	The protocol specified is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attribute: being set to an inconsistent state).	
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 cont in charge.	
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this function.	
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.	
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	
VI_ERROR_TMO	Timeout expired before function completed.	

See Also

<u>VI_ATTR_TRIG_ID</u> attribute. Set this attribute to the trigger mechanism/ trigger line to use.

<u>VI_EVENT_TRIGGER</u> 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. You should assert SYSRESET (also known as HARD RESET in the VXI specification) *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	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
line	IN	ViUInt16	Specifies the utility bus signal to assert. This can be: 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	The specified utility bus signal was successfully asserted to the device.
Error Codes	Description
VI_ERROR_INV_LINE	The value specified by the line parameter is invalid.
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 <i>vi</i> does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_SUCCESS	Operation completed successfully.

See Also

BACKPLANE resource description
viBufRead

Syntax

```
viBufRead(ViSession vi, ViPBuf buf, ViUInt32 count,
ViPUInt32 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	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
buf	OUT	ViPBuf	Represents the location of a buffer to receive data from the device.
count	IN	ViUInt32	Number of bytes to be read.
retCount	OUT	ViPUInt32	Represents the location of an integer that will be set to the 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 Codes	Description	
VI_SUCCESS	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).	
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to count	
VI_SUCCESS_TERM_CHAR	The specified termination character was read.	
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_IO	An unknown I/O error occurred during transfer.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> 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.	

See Also

viWrite, viScanf

viBufWrite

Syntax

```
viBufWrite(ViSession vi, ViBuf buf, ViUInt32 count,
ViPUInt32 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	Dir	Туре	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	ViPUInt32	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_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_IO	An unknown I/O error occurred during transfer.		
VI_ERROR_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 <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before function completed.		

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. For a session to a USB instrument, this function sends the INITIATE_CLEAR and CHECK_CLEAR_STATUS commands on the control pipe.

Parameters

Name	Dir	Туре	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_BERR	Bus error occurred during transfer.		
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_NCIC	The interface associated with the given vi is not currently the controller in charge.		
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).		
VI_ERROR_NSUP_OPER	The given vi does not support this function.		
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI ERROR TMO	Timeout expired before function completed.		

viClose

Syntax

viClose(ViSession/ViEvent/ViFindList vi);

Description

This function closes the specified resource manager session, device session, find list (returned from the <u>viFindRsrc</u> function), or event context (returned from the <u>viWaitOnEvent</u> 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

<u>Name</u> <u>Dir</u> <u>Type</u> <u>Description</u>

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

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	Operation completed successfully.	
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.	
Error Codes	Description	
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	

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	<u>Dir</u>	Туре	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 Parame	ter		
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_IO_COMPLETION			Notification that an asynchronous operation has completed.
VI_EVENT_TRIG			Notification that a hardware trigger was received from a device.
VI_EVENT_SERVICE_REQ			Notification that a device is requesting service.
VI_EVENT_CLEAR			Notification that the local controller has been sent a device clear message
VI_EVENT_EXCEPTION			Notification that an error condition has occurred during an operation invocation. (Note: the VI_QUEUE and VI_SUSPEND_HNDLR mechanisms cannot be used

	with this event.)
VI_EVENT_GPIB_CIC	Notification that the GPIB controller has gained or lost CIC (controller in charge) status.
VI_EVENT_GPIB_TALK	Notification that the GPIB controller has been addressed to talk.
VI_EVENT_GPIB_LISTEN	Notification that the GPIB controller has been addressed to listen.
VI_EVENT_PXI_INTR	Notification that a vendor- specific PXI interrupt was received from the device.
VI_EVENT_VXI_VME_SYSFAIL	Notification that the VXI/VME SYSFAIL* line has been asserted.
VI_EVENT_VXI_VME_SYSRESET	Notification that the VXI/VME SYSRESET* line has been asserted
VI_EVENT_VXI_SIGP	Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_VXI_VME_INTR	Notification that a VXIbus interrupt was received from the device.
Not supported by Agilent VISA:	
VI_EVENT_TCPIP_CONNECT	Notification that a TCP/IP connection has been made.
VI_EVENT_USB_INTR	Notification that a vendor- specific USB interrupt was received from the device.

to

to

Note: Refer to the <u>viEventHandler</u> topic and the <u>VISA Attributes</u> topics for information on the event types that are available for various VISA resource classes (e.g. INSTR, INTFC ...).

Special Values for *mechanism* **Parameter**

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

callback queue.

Disable this session from receiving the specified event(s) via the waiting queue.

VI_QUEUE

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

See Also

See the handler prototype <u>viEventHandler</u> for its parameter description, and <u>viEnableEvent</u>. Also, see <u>viInstallHandler</u> and <u>viUninstallHandler</u> 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 that 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	Dir	Туре	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 <i>eventType</i> Param	eter		
Value	_		Action Description
VI_ALL_ENABLED_EVENTS			Discard events of every type that is enabled.
The following events can be discard	led:		
Event Name	_		Description
VI_EVENT_IO_COMPLETION			Notification that an asynchronous operation has completed.
VI_EVENT_TRIG			Notification that a hardware trigger was received from a device.
VI_EVENT_SERVICE_REQ			Notification that a device is requesting service.
VI_EVENT_CLEAR			Notification that the local controller has been sent a device clear message
			Notification that an error condition has occurred during an operation invocation. (Note: the

VI_EVENT_EXCEPTION	VI_QUEUE and VI_SUSPEND_HNDLR mechanisms cannot be used with this event.)
VI_EVENT_GPIB_CIC	Notification that the GPIB controller has gained or lost CIC (controller in charge) status.
VI_EVENT_GPIB_TALK	Notification that the GPIB controller has been addressed to talk.
VI_EVENT_GPIB_LISTEN	Notification that the GPIB controller has been addressed to listen.
VI_EVENT_PXI_INTR	Notification that a vendor- specific PXI interrupt was received from the device.
VI_EVENT_VXI_VME_SYSFAIL	Notification that the VXI/VME SYSFAIL* line has been asserted.
VI_EVENT_VXI_VME_SYSRESET	Notification that the VXI/VME SYSRESET* line has been asserted
VI_EVENT_VXI_SIGP	Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_VXI_VME_INTR	Notification that a VXIbus interrupt was received from the device.
Not supported by Agilent VISA:	
VI_EVENT_TCPIP_CONNECT	Notification that a TCP/IP connection has been made.
VI_EVENT_USB_INTR	Notification that a vendor- specific USB interrupt was received from the device.

Note: Refer to the <u>viEventHandler</u> topic and the <u>VISA Attributes</u> topics for information on the event types that are available for various VISA resource classes (e.g. INSTR, INTFC ...).

Special Values for *mechanism* **Parameter**

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

VI_SUSPEND_HNDLR

Discard the specified event(s) from the callback queue.

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_EVENT	Specified event type is not supported by the resource.	
VI EDDOD INV MECH		
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 <u>viClose</u> 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	Dir	Туре	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 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.
context	IN	ViEventFilter	VI_NULL (Not used for VISA 1.0.)

Special Values for eventType Parameter

Value

VI_ALL_ENABLED_EVENTS

The following events can be enabled:

Event Name

VI_EVENT_IO_COMPLETION

VI_EVENT_TRIG

VI_EVENT_SERVICE_REQ

VI_EVENT_CLEAR

Action Description

Switch all events that were previously enabled to the callback mechanism specified in the mechanism parameter.

Description

Notification that an asynchronous operation has completed.

Notification that a hardware trigger was received from a device.

Notification that a device is requesting service.

Notification that the local controller has been sent a device

	clear message
VI_EVENT_EXCEPTION	Notification that an error condition has occurred during an operation invocation. (Note: the VI_QUEUE and VI_SUSPEND_HNDLR mechanisms cannot be used with this event.)
VI_EVENT_GPIB_CIC	Notification that the GPIB controller has gained or lost CIC (controller in charge) status.
VI_EVENT_GPIB_TALK	Notification that the GPIB controller has been addressed to talk.
VI_EVENT_GPIB_LISTEN	Notification that the GPIB controller has been addressed to listen.
VI_EVENT_PXI_INTR	Notification that a vendor-specific PXI interrupt was received from the device.
VI_EVENT_VXI_VME_SYSFAIL	Notification that the VXI/VME SYSFAIL* line has been asserted.
VI_EVENT_VXI_VME_SYSRESET	Notification that the VXI/VME SYSRESET* line has been asserted
VI_EVENT_VXI_SIGP	Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_VXI_VME_INTR	Notification that a VXIbus interrupt was received from the device.
Not supported by Agilent VISA:	
VI_EVENT_TCPIP_CONNECT	Notification that a TCP/IP connection has been made.
VI_EVENT_USB_INTR	Notification that a vendor-specific USB interrupt was received from the device.

Note: Refer to the <u>viEventHandler</u> topic and the <u>VISA Attributes</u> topics for information on the event types that are available for various VISA resource classes (e.g. INSTR, INTFC ...).

Special Values for *mechanism* **Parameter**

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).

Value	Action Description
VI_HNDLR	Enable this session to receive the specified event via a callback handler, which must have already been installed via <u>viInstallHandler</u> .
VI_QUEUE	Enable this session to receive the specified event via the waiting queue. Events must be retrieved manually via the <u>viWaitOnEvent</u> function.
VI_SUSPEND_HNDLR	Enable this session to receive the specified event via a callback queue. Events will not be delivered to the session until <u>viEnableEvent</u> is invoked again with the VI_HNDLR 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_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_INV_CONTEXT	Specified event context is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_MECH	The specified mechanism is not supported for the given event type.

See Also

See the handler prototype <u>viEventHandler</u> for its parameter description and <u>viDisableEvent</u>. Also, see the <u>viInstallHandler</u> and <u>viUninstallHandler</u> 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 that VISA requires. Visual Studio C++ defaults to *cdecl* that 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 <u>viClose</u> 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.

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 <u>viReadSTB</u> function to read the status byte of the service request.

Instrument Control (INSTR) Resource Events

Event Name	Attributes	Data Type	Range
		<u></u>	
VI_EVENT_PXI_INTR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_PXI_IN
VI_EVENT_SERVICE_REQ	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVIC
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE VI_ATTR_SIGP_STATUS_ID	ViEventType ViUInt16	VI_EVENT_VXI_S] 0 to FFFFh
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIG VI_TRIG_TTL0 to V VI_TRIG_ECL0 to V
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_COI N/A N/A N/A 0 to FFFFFFFh N/A
VI_EVENT_VXI_VME_INTR	VI_ATTR_EVENT_TYPE VI_ATTR_INTR_STATUS_ID VI_ATTR_RECV_INTR_LEVEL	ViEventType ViUInt32 ViInt16	VI_EVENT_VXI_V 0 to FFFFFFFh 1 to 7, VI_UNKNO\
Not supported by Agilent VISA:			
VI_EVENT_USB_INTR	VI_ATTR_EVENT_TYPE VI_ATTR_USB_RECV_INTR_SIZE VI_ATTR_USB_RECV_INTR_DATA VI_ATTR_STATUS	ViEventType ViUInt16 ViBuf ViStatus	VI_EVENT_USB_I 0 to FFFFh N/A N/A

Memory Access (MEMACC) Resource Events

Event Name	Attributes	Data — <u>Type</u>	Range	
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_CO N/A N/A N/A 0 to FFFFFFFh N/A	

GPIB Bus Interface (INTFC) Resource Events

Event Name	Attributes	Data	Range
Event Name	Attributes	Dala	Range

Туре

VI_EVENT_GPIB_CIC	VI_ATTR_EVENT_TYPE VI_ATTR_GPIB_RECV_CIC_STATE		VI_EVENT_GPIB_(VI_TRUE VI_FALSE
VI_EVENT_GPIB_TALK	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_7
VI_EVENT_GPIB_LISTEN	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_I
VI_EVENT_CLEAR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_CLEAR
VI_EVENT_TRIGGER	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIGG VI_TRIG_SW
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_CO! N/A N/A 0 to FFFFFFFh N/A

VXI Mainframe Backplane (BACKPLANE) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIG VI_TRIG_TTL0 to V VI_TRIG_ECL0 to V
VI_EVENT_VXI_VME_SYSFAIL	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VI
VI_EVENT_VXI_VME_SYSRESET	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VI

TCPIP Socket (SOCKET) Resource Events

Event Name	Attributes	Data — Type	Range	
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_CO! N/A N/A 0 to FFFFFFFh N/A	

Parameters

Name	Dir	Туре	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 queue flushed successfully.
VI_SUCCESS_NCHAIN	Event handled successfully. Do not invoke any other handlers on this session for this event.

See Also

See <u>Programming with VISA</u> in the *Agilent VISA User's Guide* 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 <u>viFindRsrc</u>. The list is referenced by the handle that was returned by **viFindRsrc**.

Note: If you use the Agilent Connection Expert to put a resource into the *ignored* state, **viFindRsrc** and **viFindNext** will not find that resource. If an interface is in the *ignored* state, **viFindRsrc** and **viFindNext** will find neither the interface nor instruments on that interface.

Connection Expert's **Address check** property on an instrument also affects the operation of **viFindRsrc** and **viFindNext** for that instrument. <u>Click here</u> for more information on this interaction.

Parameters

Name	<u>Dir</u>	Туре	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 <u>viOpen</u> 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	The given <i>findList</i> does not support this function.	
VI_ERROR_RSRC_NFOUND	There are no more matches.	

See Also

viFindRsrc Address Check, viFindRsrc, and viOpen

viFindRsrc

Syntax

viFindRsrc(ViSession sesn, ViString expr, ViPFindList findList, ViPUInt32 retcnt, ViPRsrc 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.

Note: If you use the Agilent Connection Expert to put a resource into the *ignored* state, **viFindRsrc** and **viFindNext** will not find that resource. If an interface is in the *ignored* state, **viFindRsrc** and **viFindNext** will find neither the interface nor instruments on that interface.

Connection Expert's **Address check** property on an instrument also affects the operation of **viFindRsrc** and **viFindNext** for that instrument. <u>Click here</u> for more information on this interaction.

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 <u>viFindNext</u>. When this handle is no longer needed, it should be passed to <u>viClose</u>.

The search criteria specified in the *expr* parameter have 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.

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.

Special Character	Meaning		
&&	Logical AND		
	Logical OR		
!	Logical negation (NOT)		
0	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 <u>viFindRsrc</u> 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 <u>viClose</u> 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 <u>viParseRsrc</u>, <u>viParseRsrcEx</u>, and <u>viOpen</u>. However, not all resource strings that can be parsed or opened have to be findable.

Parameters

Name	Dir	Туре	Description
sesn	IN	ViSession	Resource Manager session (should always be the Default Resource Manager for VISA returned from <u>viOpenDefaultRM</u>).
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	ViPFindList	Returns a handle identifying this search session. This handle will be used as an input in <u>viFindNext</u> .
retcnt	OUT	ViPUInt32	Number of matches.
instrDesc	OUT	ViPRsrc	Returns a string identifying the location of a device. Strings can then be passed to <u>viOpen</u> to establish a session to the given device.

Description String for *expr* **Parameter**

Expression
GPIB[0-9]*::?*INSTR
PXI?*INSTR
VXI?*INSTR
GPIB-VXI?*INSTR
GPIB?*INSTR
?*VXI[0-9]*::?*INSTR
ASRL[0-9]*::?*INSTR
?*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_EXPR	Invalid expression specified for search.	
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 <i>sesn</i> does not support this function.	
VI_ERROR_RSRC_NFOUND	Specified expression does not match any devices.	

See Also

<u>viFindNext, viClose</u> <u>Address Check, viFindRsrc, and viOpen</u>

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:

Mask Values	Interpretation	
VI_IO_IN_BUF	Discard receive buffer contents (same as VI_IO_IN_BUF_DISCARD).	
VI_IO_IN_BUF_DISCARD	Discard receive 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).	
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 message="" response="">. (See the IEEE 488.2 standard.)</terminated>	
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).	

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

<u>Name</u>	<u>Dir</u>	Туре	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 "Description" section for details.)

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_MASK	The specified mask does not specify a valid flush function on read/write resource.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_IO	Could not perform read/write function because of I/O error.	
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	The read/write function was aborted because timeout expired while function was in progress.	

See Also

<u>viSetBuf</u>

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	Dir	Туре	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,
ViPUInt32 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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session
	111	VISESSION	
buf	IN	ViBut	Buffer containing valid GPIB commands.
count	IN	ViUInt32	Number of bytes to be written.
retCount	OUT	ViPUInt32	Number of bytes actually transferred.

Special Value for *retcnt* **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_INV_SETUP	Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_IO	An unknown I/O error occurred during transfer.		
VI_ERROR_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_NSUP_OPER	The given <i>vi</i> does not support this operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before operation completed.		

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	Dir	Туре	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			
Value			Action Description
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			Deassert ATN line.
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.
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_MODE	The value specified by the mode parameter is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.	
VI_ERROR_NSUP_MODE	The specified mode is not supported by this VISA implementation.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> 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.	

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	Dir	Туре	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_ADDRESS_GTL			Send the Go To Local command (GTL) to this device.
VI_GPIB_REN_ASSERT			Assert REN line.
VI_GPIB_REN_ASSERT_ADDRESS_LLO			Address this device and send it LLO, putting it in RWLS.
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.
VI_GPIB_REN_DEASSERT			Deassert REN line.
VI_GPIB_REN_DEASSERT_GTL			Send the Go To Local command (GTL) to this device and deassert REN 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	Resource attribute retrieved successfully.	
Error Codes	Description	
VI_ERROR_INV_MODE	The value specified by the mode parameter is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
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_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_NSYS_CNTLR	The interface associated with this session is not the system controller.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

viGpibPassControl

Syntax

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

Description

Tells 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	<u>Dir</u>	Туре	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_OBJECTON	The given session or object reference is invalid (both are the same value).	
VI_ERROR_IO	An unknown I/O error occurred during transfer.	
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_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	
VI_ERROR_TMO	Timeout expired before operation completed.	

See Also

INTFC resource description

viGpibSendIFC

Syntax

viGpibSendIFC(ViSession vi);

Description

Pulse the interface clear line (IFC) for at least 100 mseconds. 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	Dir	Туре	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 <i>vi</i> does not support this operation.	
VI_ERROR_NSYS_CNTLR	The interface associated with this session is not the system controller.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

INTFC resource description

viIn8Ex, viIn16Ex, viIn32Ex, and viIn64Ex

Syntax

- viIn8Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViPUInt8 val8);
- viIn16Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViPUInt16 val16);
- viIn32Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViPUInt32 val32);
- viIn64Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViPUInt64 val64);

Description

These VISA 4.0 operations, by using the specified address space, read in 8, 16, 32 or 64 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, 32-bit or 64-bit value from the specified memory space (assigned memory base + offset). This function takes the 8-bit, 16-bit, 32-bit, or 64-bit value from the address space pointed to by *space*. The offset must be a valid memory address in the space.

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.
VI_A64_SPACE	Address the A64 address space of VXI/MXI bus.
VI_PXI_CFG_SPACE	Address the PCI configuration space.
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.

The high-level operations <u>viIn8Ex</u>, <u>viIn16Ex</u>, <u>viIn32Ex</u> and <u>viIn64Ex</u> operate successfully independently from the low-level operations (<u>viMapAddressEx</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u> and <u>viPoke64</u>). The high-level and low-level operations are independent regardless of the configured state of the hardware that is used to perform memory accesses.

The offset specified in the **viIn8Ex**, **viIn16Ex**, **viIn32Ex**, and **viIn64Ex** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified. For a MEMACC resource, the offset parameter specifies an absolute address.

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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space.

Parameters

Name	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
space	IN	ViUInt16	Specifies the address space. (See the following table.)
offset64	IN	ViBusAddress64	Offset (in bytes) of the memory to read from.
val8, val16, val32, or val64	OUT	ViPUInt8, ViPUInt16, ViPUInt32, or ViPUInt64	Data read from bus (8 bits for viIn8 , 16 bits for viIn16 , 32 bits for viIn32 , or 64 bits for viIn64).

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_BERR	Bus error occurred during transfer.	
VI_ERROR_INV_OFFSET	Invalid offset specified.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SPACE	Invalid address space specified.	
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.	
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

viOut8Ex, viOut16Ex, viOut32Ex, viOut64Ex, viPeek8, viPeek16, viPeek32, viPeek64, viMoveIn8Ex, viMoveIn16Ex, viMoveIn32Ex, viMoveIn64Ex

viIn8, viIn16, viIn32, and viIn64

Syntax

- viIn8(ViSession vi, ViUInt16 space, ViBusAddress offse
 ViPUInt8 val8);
- viIn16(ViSession vi, ViUInt16 space, ViBusAddress offs ViPUInt16 val16);
- viIn32(ViSession vi, ViUInt16 space, ViBusAddress offs ViPUInt32 val32);
- viIn64(ViSession vi, ViUInt16 space, ViBusAddress offs ViPUInt64 val64); [VISA 4.0 and later]

Description

This operation, by using the specified address space, reads in 8, 16, 32 or 64 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, 32-bit or 64-bit value from the specified memory space (assigned memory base + offset). This function takes the 8-bit, 16-bit, 32-bit, or 64-bit value from the address space pointed to by *space*. The offset must be a valid memory address in the space.

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
	Address the A1C address space of VVI/AVI hus
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.
VI_A64_SPACE	Address the A64 address space of VXI/MXI bus.
VI_PXI_CFG_SPACE	Address the PCI configuration space.
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.

The high-level operations <u>viIn8</u>, <u>viIn16</u>, <u>viIn32</u> and <u>viIn64</u> operate independently from the low-level operations (<u>viMapAddress</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u> and <u>viPoke64</u>). The highlevel and low-level operations are independent 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, viIn32**, and **viIn64** 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space.

Parameters

Name	Dir	Туре	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, val32, or val64	OUT	ViPUInt8, ViPUInt16, ViPUInt32, or ViPUInt64	Data read from bus (8 bits for viIn8 , 16 bits for viIn16 , 32 bits for viIn32 , or 64 bits for viIn64).

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_BERR	Bus error occurred during transfer.	
VI_ERROR_INV_OFFSET	Invalid offset specified.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SPACE	Invalid address space specified.	
VI_ERROR_NSUP_ALIGN_OFFSET	The specified offset is not properly aligned for the access width of the operation.	
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

viOut8, viOut16, viOut32, viOut64, viPeek8, viPeek16, viPeek32, viPeek64, viMoveIn8, viMoveIn16, viMoveIn32, viMoveIn64
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	<u>Dir</u>	Туре	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 are valid:			
Event Name			Description
VI_EVENT_IO_COMPLETION			Notification that an asynchronous operation has completed.
VI_EVENT_TRIG			Notification that a hardware trigger was received from a device.
VI_EVENT_SERVICE_REQ			Notification that a device is requesting service.
VI_EVENT_CLEAR			Notification that the local controller has been sent a device clear message
VI_EVENT_EXCEPTION			Notification that an error condition has occurred during an operation invocation. (Note: the VI_QUEUE and VI_SUSPEND_HNDLR mechanisms cannot be used with this event.)
VI_EVENT_GPIB_CIC			Notification that the GPIB controller has gained or lost CIC (controller in charge) status.
VI_EVENT_GPIB_TALK			Notification that the GPIB controller has been addressed to talk.
VI_EVENT_GPIB_LISTEN			Notification that the GPIB controller has been addressed to listen.
VI_EVENT_PXI_INTR			Notification that a vendor-specific PXI interrupt was received from the device.
VI_EVENT_VXI_VME_SYSFAIL			Notification that the VXI/VME SYSFAIL* line has been asserted.
VI_EVENT_VXI_VME_SYSRESET			Notification that the VXI/VME SYSRESET* line has been asserted
VI_EVENT_VXI_SIGP			Notification that a VXI signal or VXI interrupt has been received from a device.
VI_EVENT_VXI_VME_INTR			Notification that a VXIbus interrupt was received

from the device.

Not supported by Agilent VISA:

VI_EVENT_TCPIP_CONNECT

VI_EVENT_USB_INTR

Notification that a TCP/IP connection has been made.

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

Note: Refer to the *viEventHandler* topic and the *VISA Attributes* topics for information on the event types that are available for various VISA resource classes (e.g. INSTR, INTFC ...).

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_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.	
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_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	

See Also

<u>viEventHandler</u>

viLock

Syntax

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

Note: Shared locks ARE supported by HiSLIP devices but are NOT supported on other types of 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 <u>viUnlock</u> 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.

Parameters

Name	Dir	Туре	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.
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 <i>lockType</i> 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 <i>accessKey</i> for the session, or the session can suggest an <i>accessKey</i> to use for the shared lock. See "Description" for more details.
accessKey	OUT	ViKeyId	This parameter should be set to VI_NULL when <i>lockType</i> 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 <i>accessKey</i> 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 specified access mode was successfully acquired, and this session has nested shared locks.
Error Codes	Description
VI_ERROR_INV_ACCESS_KEY	The <i>requestedKey</i> value passed is not a valid access key to the specified resource.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given <i>vi</i> 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_TMO	The specified type of lock could not be obtained within the specified <i>timeout</i> period.

See Also

<u>viUnlock</u>. For more information on locking, see <u>Programming with VISA</u> in the *Agilent VISA User's Guide*.

viMapAddress

Syntax

viMapAddress(ViSession vi, ViUInt16mapSpace, ViBusAddress mapBase , ViBusSizemapSize, ViBooleanaccess, ViAddrsuggested, ViPAddraddress);

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.	
VI_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.	

• 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space. • When calling the viMapAddress function on a PXI session, the maximum value for the *mapSize* parameter is 1048576 (0x100000) even if the map space being referenced is larger than this value. If you need access to a larger memory block than this, there are two ways to work around this limit:

1. Use viMoveIn, viMoveOut, or viMove. These functions are not limited to the 0x100000 byte limit.

2. Use viMapAddress / viUnmapAddress multiple times to map individual blocks of memory that are less than 0x100000 bytes and access each block in sequence.

Parameters

Name	<u>Dir</u>	Туре	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 <i>suggested</i> 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 <i>suggested</i> .
address	OUT	ViPAddr	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_ALLOC	Unable to allocate window of at least the requested size.		
VI_ERROR_INV_ACC_MODE	Invalid access mode.		
VI_ERROR_INV_OFFSET	Invalid offset specified.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_INV_SETUP	Unable to start operation because the setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_INV_SIZE	Invalid size of window specified.		
VI_ERROR_INV_SPACE	Invalid <i>mapSpace</i> specified.		
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.		
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	<i>viMapAddress</i> could not acquire resource or perform mapping before the timer expired.		
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.		

See Also

viUnmapAddress

viMapAddressEx

Note: This function is not currently supported in Agilent VISA.

Syntax

viMapAddressEx(ViSession vi, ViUInt16 mapSpace, ViBusAddress64 mapBase64, ViBusSize mapSize, ViBoole access, ViAddr suggested, ViPAddr address);

Description

This VISA 4.0 (and later) 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 VAI/MAI 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.
VI_A64_SPACE	Address the A64 address space of VXI/MXI bus.
VI_PXI_CFG_SPACE	Address the PCI configuration space.
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.

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 **viMapAddressEx** 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, *mapBase64* specifies the 64-bit offset from the logical address base address of the VXI device specified. If *mapSpace* specifies VI_A24_SPACE, VI_A32_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, A32, or A64 space.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
mapSpace	IN	ViUInt16	Specifies the address space to map.
mapBase64	IN	ViBusAddress64	64-bit 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 <i>suggested</i> 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 <i>suggested</i> .
address	OUT	ViPAddr	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_ALLOC	Unable to allocate window of at least the requested size.		
VI_ERROR_INV_ACC_MODE	Invalid access mode.		
VI_ERROR_INV_OFFSET	Invalid offset specified.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_INV_SETUP	Unable to start operation because the setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_INV_SIZE	Invalid size of window specified.		
VI_ERROR_INV_SPACE	Invalid <i>mapSpace</i> specified.		
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.		
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	<i>viMapAddress</i> could not acquire resource or perform mapping before the timer expired.		
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.		

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 (<u>BACKPLANE</u>) 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	Dir	Туре	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.
	IN		
mode	IN	ViUInt16	Specifies the trigger mapping mode. This should always be VI_NULL for VISA 3.0 and later.

Special Values for *trgSrc* and *trigDest* Parameters

Value	Action Description
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.
VI_TRIG_TTL0 - VI_TRIG_TTL7	Map the specified VXI TTL trigger line.

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	Operation completed successfully.	
VI_SUCCESS_TRIG_MAPPED	The path from <i>trigSrc</i> to <i>trigDest</i> is already mapped.	
Error Codes	Description	
VI_ERROR_INV_LINE	One of the specified lines (trigSrc or trigDest) is invalid.	
VI_ERROR_INV_MODE	The value specified by the mode parameter is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_LINE_IN_USE	One of the specified lines (<i>trigSrc</i> or <i>trigDest</i>) is currently in use.	
VI_ERROR_NSUP_LINE	One of the specified lines (<i>trigSrc</i> or <i>trigDest</i>) is not supported by this VISA implementation.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	
VI_ERROR_TMO	viMapAddress could not acquire resource or perform mapping before the timer expired.	

See Also

BACKPLANE Resource Description

viMemAlloc

Syntax

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

Description

Note: viMemAlloc is implemented for PXI MEMACC resources only. It is not implemented for VXI and VXI-GPIB MEMACC resources.

The offset returned for a PXI MEMACC resource is an absolute physical address of a contiguous block of memory which has been allocated by the operating system. The memory block is also locked, meaning it will not be swapped out. A common use for this memory is to do user-mode DMA to or from a PXI device..

Both the high-level memory access functions (viInXX, viOutXX, viMoveXX) and the low-level memory access functions (viMapAddress, viPeekXX, viPokeXX, viUnmapAddress) can be used on MEMACC sessions to access the allocated memory.

Parameters

Name	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
size	IN	ViBusSize	Specifies the size of the allocation.
offset	OUT	ViPBusAddress	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	Operation completed successfully.	
Error Codes	Description	
VI_ERROR_ALLOC	Unable to allocate shared memory block of the requested size.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SIZE	Invalid size specified.	
VI_ERROR_MEM_NSHARED	The device does not export any memory.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

<u>viMemFree</u>

viMemAllocEx

Syntax

viMemAllocEx(ViSession vi, ViBusSize
 size, ViBusAddress64 offset64); [VISA 4.0 and later]

Description

Note: viMemAllocEx is implemented for PXI MEMACC resources only. It is not implemented for VXI and VXI-GPIB MEMACC resources.

The offset returned for a PXI MEMACC resource is an absolute physical address of a contiguous block of memory which has been allocated by the operating system. The memory block is also locked, meaning it will not be swapped out. A common use for this memory is to do user-mode DMA to or from a PXI device..

Both the high-level memory access functions (viInXX, viOutXX, viMoveXX) and the low-level memory access functions (viMapAddress, viPeekXX, viPokeXX, viUnmapAddress) can be used on MEMACC sessions to access the allocated memory.
Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
size	IN	ViBusSize	Specifies the size of the allocation.
offset64	OUT	ViBusAddress64	Returns the 64-bit 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	Operation completed successfully.	
Error Codes	Description	
VI_ERROR_ALLOC	Unable to allocate shared memory block of the requested size.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SIZE	Invalid size specified.	
VI_ERROR_MEM_NSHARED	The device does not export any memory.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

<u>viMemFreeEx</u>

viMemFree

Syntax

viMemFree(ViSession vi, ViBusAddress offset);

Description

Note: viMemFree is implemented for PXI MEMACC resources only. It is not implemented for VXI and VXI-GPIB MEMACC resources.

This function frees the memory previously allocated using viMemAlloc.

Both the high-level memory access functions (viInXX, viOutXX, viMoveXX) and the low-level memory access functions (viMapAddress, viPeekXX, viPokeXX, viUnmapAddress) can be used on MEMACC sessions to access the allocated memory.

Parameters

<u>Name</u>	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
offset	IN	ViBusAddress	Specifies the memory previously allocated with <u>viMemAlloc</u> .

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_OFFSET	Invalid offset specified.	
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 <i>vi</i> does not support this operation.	
VI_ERROR_WINDOW_MAPPED	The specified offset is currently in use by viMapAddress .	

See Also

<u>viMemAlloc</u>

viMemFreeEx

Syntax

viMemFreeEx(ViSession vi, ViBusAddress64 offset64);
[VISA 4.0 and later]

Description

Note: viMemFreeEx is implemented for PXI MEMACC resources only. It is not implemented for VXI and VXI-GPIB MEMACC resources.

This VISA 4.0 (and later) function frees the memory previously allocated using viMemAllocEx.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
offset64	IN	ViBusAddress64	Specifies the memory previously allocated with <u>viMemAllocEx</u> .

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_OFFSET	Invalid offset specified.	
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 <i>vi</i> does not support this operation.	
VI_ERROR_WINDOW_NMAPPED	The specified offset is currently in use by viMapAddress .	

See Also

<u>viMemAllocEx</u>

viMove

Syntax

viMove(ViSession vi, ViUInt16 srcSpace, ViBusAddress srcOffset, viUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress destOffset, ViUInt16 destWidth, ViBusSi 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 <u>INSTR</u> or <u>MEMACC</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_LOCAL_SPACE	Address the process-local memory (using virtual address).	
VI_OPAQUE_SPACE	Addresses potentially volatile data (using a virtual address).	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.	

Valid entries for specifying address space:

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 (<u>viMapAddress</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u>, and <u>viPoke64</u>). The high-level and low-level operations are independent 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	<u>Dir</u>	Туре	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	Offset of the starting address or register to which to write
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 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_BERR	Bus Error occurred during transfer.	
VI_ERROR_INV_LENGTH	Invalid length specified.	
VI_ERROR_INV_OFFSET	Invalid source or destination offset specified.	
VI_ERROR_INV_SESSION VI_ERROR_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).	
VI_ERROR_INV_SPACE	Invalid source or destination address specified.	
VI_ERROR_INV_WIDTH	Invalid source or destination width specified.	
VI_ERROR_NSUP_ALIGH_OFFSET	The specified offset is not properly aligned for the access width of the operation.	
VI_ERROR_NSUP_OFFSET	Specified source or destination offset is not accessible from this hardware.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_NSUP_VAR_WIDTH	Cannot support source and destination widths that are different.	
VI_ERROR_NSUP_WIDTH	Specified width is not supported.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

viMoveAsync. Also, see the <u>MEMACC</u> Resource description.

viMoveEx

Syntax

viMoveEx(ViSession vi, ViUInt16 srcSpace, ViBusAddress srcOffset64, viUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress64 destOffset64, ViUInt16 destWidth, ViBusSize length);

Description

This VISA 4.0 (and later) 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 <u>INSTR</u> or <u>MEMACC</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_LOCAL_SPACE	Address the process-local memory (using virtual address).	
VI_OPAQUE_SPACE	Addresses potentially volatile data (using a virtual address).	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.	

Valid entries for specifying address space:

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.

VI_WIDTH_64 Performs a 64-bit (D64) transfer.

The high-level operation **viMoveEx** operates successfully independently from the low-level operations (<u>viMapAddress</u>, <u>viMapAddressEx</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u> and <u>viPoke64</u>). The high-level and low-level operations are independent regardless of the configured state of the hardware that is used to perform memory accesses.

The length specified in the **viMoveEx** 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 **viMoveEx** operation for a MEMACC resource are absolute addresses.

Parameters

Name Dir		Туре	Description	
vi	IN	ViSession	Unique logical identifier to a session.	
srcSpace	IN	ViUInt16	Specifies the address space of the source.	
srcOffset64	IN	ViBusAddress64	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.	
destOffset64	IN	ViBusAddress64	Offset of the starting address or register to which to write.	
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 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_BERR	Bus Error occurred during transfer.	
VI_ERROR_INV_LENGTH	Invalid length specified.	
VI_ERROR_INV_OFFSET	Invalid source or destination offset specified.	
VI_ERROR_INV_SESSION VI_ERROR_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).	
VI_ERROR_INV_SPACE	Invalid source or destination address specified.	
VI_ERROR_INV_WIDTH	Invalid source or destination width specified.	
VI_ERROR_NSUP_ALIGH_OFFSET	The specified offset is not properly aligned for the access width of the operation.	
VI_ERROR_NSUP_OFFSET	Specified source or destination offset is not accessible from this hardware.	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this operation.	
VI_ERROR_NSUP_VAR_WIDTH	Cannot support source and destination widths that are different.	
VI_ERROR_NSUP_WIDTH	Specified width is not supported.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

viMoveAsyncEx. Also, see the MEMACC Resource description.

viMoveAsync

Syntax

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

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 <u>viTerminate</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.		
VI_LOCAL_SPACE	Address the process-local memory (using virtual address).		
VI_OPAQUE_SPACE	Addresses potentially volatile data (using a virtual address).		
VI_PXI_CFG_SPACE	Address the PCI configuration space.		
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.		
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.		

Valid entries for specifying address space:

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.	
VI_WIDTH_64	Performs a 64-bit (D64) transfer.	

Programming Tip: Performing multiple asynchronous operations

simultaneously: The VISA asynchronous functions viMoveAsync, viReadAsync and viWriteAsync initiate I/O operations to a device on a separate thread which allows the main thread to continue without blocking when doing I/O. VISA allows you to initiate multiple simultaneous asynchronous operations on a single VISA session, but the Agilent IO Libraries Suite allows only a single thread at a time from a given session to access the device. To perform multiple asynchronous operations simultaneously, you can work around this limitation by opening multiple sessions to the device and doing one VISA asynchronous call on each session.

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**.

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	Dir	Туре	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	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	ViPJobId	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 fo	or jobId	Parameter	
Value			Action Description
VI_NULL			Operation does 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	Operation completed successfully.		
VI_SUCCESS_SYNC	Operation performed 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 <i>vi</i> does not support this operation.		
VI_ERROR_QUEUE	Unable to queue move operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_IN_PROGRESS	Unable to start a new asynchronous operation while another asynchronous operation is in progress.		

See Also

viMove. Also, see INSTR and MEMACC Resource descriptions.

viMoveAsyncEx

Note: This function is not currently supported in Agilent VISA.

Syntax

viMoveAsyncEx(ViSession vi, ViUInt16 srcSpace, ViBusAddress64 srcOffset64, ViUInt16 srcWidth, ViUInt16 destSpace, ViBusAddress64 destOffset64, ViUInt16 destWidth, ViBusSize length, ViPJobId jobId

Description

This VISA 4.0 (and later) 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 <u>viTerminate</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.		
VI_LOCAL_SPACE	Address the process-local memory (using virtual address).		
VI_OPAQUE_SPACE	Addresses potentially volatile data (using a virtual address).		
VI_PXI_CFG_SPACE	Address the PCI configuration space.		
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.		
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.		

Valid entries for specifying address space:

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 VI_WIDTH_64	Performs a 32-bit (D32) transfer. Performs a 64-bit (D64) transfer.		

Since an asynchronous I/O request could complete before the **viMoveAsyncEx** 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 **viMoveAsyncEx** 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	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
srcSpace	IN	Vil Upt16	Specifies the address space of the source
sicopace	IIN	violinio	CA his offers of the starting of these surgerists from
srcOffset64	IN	ViBusAddress64	which to read.
srcWidth	IN	ViUInt16	Specifies the data width of the source.
destSpace	IN	ViUInt16	Specifies the address space of the destination.
destOffset64	IN	ViBusAddress64	64-bit 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	ViPJobId	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 <i>j</i>	obId Par	ameter	
Value			Action Description
VI_NULL			Operation does 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	Operation completed successfully.	
VI_SUCCESS_SYNC	Operation performed 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 <i>vi</i> does not support this operation.	
VI_ERROR_QUEUE	Unable to queue move operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	
VI_ERROR_IN_PROGRESS	Unable to start a new asynchronous operation while another asynchronous operation is in progress.	

See Also

viMoveEx. Also, see INSTR and MEMACC Resource descriptions.

viMoveIn8, viMoveIn16, viMoveIn32, and viMoveIn64

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);

viMoveIn64(ViSession vi, ViUInt16 space, ViBusAddress
offset, ViBusSize length, ViAUInt64 buf64); [VISA 4
and later]

Description

This function moves an 8-bit, 16-bit, 32-bit, or 64-bit block of data from the specified memory space (assigned *memory base* + *offset*) to local memory. This function reads the 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddress</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI_PXI_ALLOC_SPACE	Access physical locally allocated memory.	

These 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 <u>viIn8</u>, <u>viIn16</u>, <u>viIn32</u>, and <u>viIn64</u> operate independently from the low-level operations (<u>viMapAddress</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u>, and <u>viPoke64</u>).

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

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

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access SHALL accept the following value for the space parameter: VI_PXI_ALLOC_SPACE.

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, VI_A32_SPACE, or VI_A64_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, A32, or A64 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	<u>Dir</u>	Туре	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 , 32 bits for viMoveIn32 , and 64 bits for viMoveIn64 .
buf8, buf16, buf32, or buf64	OUT	ViAUInt8, ViAUInt16, ViAUInt32, or ViAUInt64	Data read from bus (8 bits for viMoveIn8 , 16 bits for viMoveIn16 , 32 bits for viMoveIn32 , and 64 bits for viMoveIn64).

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

See Also

viMoveOut8, viMoveOut16, viMoveOut32, viMoveOut64, viIn8, viIn16, viIn32, viIn64

viMoveIn8Ex, viMoveIn16Ex, viMoveIn32Ex, and viMoveIn64Ex

Syntax

viMoveIn8Ex(ViSession vi, ViUInt16 space, ViBusAddress offset64, ViBusSize length, ViAUInt8 buf8); viMoveIn16Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt16 buf16); viMoveIn32Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt32 buf32); viMoveIn64Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt64 buf64);

Description

This VISA 4.0 (and later) function moves an 8-bit, 16-bit, 32-bit, or 64-bit block of data from the specified memory space (assigned *memory base* + *offset*) to local memory. This function reads the 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddress</u> 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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI PXI ALLOC SPACE	Access physical locally allocated memory.	

The <u>viMoveInxEx</u> 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 <u>viIn8Ex</u>, <u>viIn16Ex</u>, <u>viIn32Ex</u>, and <u>viIn64Ex</u> operate independently from the low-level operations (<u>viMapAddressEx</u>, <u>viPeek8</u>, <u>viPeek16</u>, <u>viPeek32</u>, <u>viPeek64</u>, <u>viPoke8</u>, <u>viPoke16</u>, <u>viPoke32</u>, and <u>viPoke64</u>).

The high-level and low-level operations are independent 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.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access SHALL accept the following value for the space parameter: VI_PXI_ALLOC_SPACE.

The 64-bit offset specified in the **viMoveIn8Ex**, **viMoveIn16Ex**, **viMoveIn32Ex**, **and viMoveIn64Ex** 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
space	IN	ViUInt16	Specifies the address space. (See the following table.)
offset64	IN	ViBusAddress64	64-bit 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 , 32 bits for viMoveIn32 , and 64 bits for viMoveIn64 .
buf8, buf16, buf32, or buf64	OUT	ViAUInt8, ViAUInt16, ViAUInt32, or ViAUInt64	Data read from bus (8 bits for viMoveIn8 , 16 bits for viMoveIn16 , 32 bits for viMoveIn32 , and 64 bits for viMoveIn64).

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

See Also

viMoveOut8Ex, viMoveOut16Ex, viMoveOut32Ex, viMoveOut64Ex, viIn8Ex, viIn16Ex, viIn32Ex, viIn64Ex

viMoveOut8, viMoveOut16, viMoveOut32, and viMoveOut64

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);

viMoveOut64(ViSession vi, ViUInt16 space, ViBusAddress
offset, ViBusSize length, ViAUInt64 buf64); [VISA 4.0 and
later]

Description

This function moves an 8-bit, 16-bit, 32-bit, or 64-bit block of data from local memory to the specified memory space (assigned memory base + *offset*). This function writes the 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddress</u> to be called prior to its invocation.

If the *ViSession* parameter (*vi*) refers to an <u>INSTR</u> session, the offset parameters specify relative offsets from the start of the instrument's address space. If the *ViSession* parameter (*vi*) refers to a <u>MEMACC</u> 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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI PXI ALLOC SPACE	Access physical locally allocated memory.	

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).

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

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access SHALL accept the following value for the space parameter: VI_PXI_ALLOC_SPACE.

The offset specified in the **viMoveOut8**, **viMoveOut16**, viMoveOut32, and viMoveOut64 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
space	IN	ViUInt16	Specifies the address space. (See the following table.)
offset64	IN	ViBusAddress64	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 , 32 bits for viMoveOut32 or 64 bits for viMoveOut64 .
buf8, buf16, buf32, or buf64	IN	ViAUInt8, ViAUInt16, ViAUInt32, ViAUInt64	Data written to bus (8 bits for viMoveOut8 , 16 bits for viMoveOut16 , 32 bits for viMoveOut32 , and 64 bits for viMoveOut64).

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

See Also

viMoveIn8, viMoveIn16, viMoveIn32, viMoveIn64, viOut8, viOut16, viOut32, viOut64

viMoveOut8Ex, viMoveOut16Ex, viMoveOut32Ex, and viMoveOut64Ex

Syntax

viMoveOut8Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt8 buf8); viMoveOut16Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt16 buf16); viMoveOut32Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt32 buf32); viMoveOut64Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViBusSize length, ViAUInt64 buf32);

Description

This VISA 4.0 (and later) function moves an 8-bit, 16-bit, 32-bit, or 64-bit block of data from local memory to the specified memory space (assigned memory base + *offset*). This function writes the 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddressEx</u> to be called prior to its invocation.

If the *ViSession* parameter (*vi*) refers to an <u>INSTR</u> session, the offset parameters specify relative offsets from the start of the instrument's address space. If the *ViSession* parameter (*vi*) refers to a <u>MEMACC</u> 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_A64_SPACE	Address the A64 address space of VXI/MXI bus.	
VI_PXI_CFG_SPACE	Address the PCI configuration space.	
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.	
VI PXI ALLOC SPACE	Access physical locally allocated memory.	

The **viMoveOutnEx** 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).

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

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access SHALL accept the following value for

the space parameter: VI_PXI_ALLOC_SPACE.

The offset specified in the **viMoveOut8Ex**, **viMoveOut16Ex**, **viMoveOut32Ex**, and **viMoveOut64Ex** 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space.

The length specified in the **viMoveOutnEx** 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 **viMoveOutnEx** 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	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
space	IN	ViUInt16	Specifies the address space. (See the following table.)
offset64	IN	ViBusAddress64	64-bit 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 viMoveOut8Ex , 16 bits for viMoveOut16Ex , 32 bits for viMoveOut32Ex , or 64-bit for viMoveOut64Ex .
buf8, buf16, buf32, or buf64	IN	ViAUInt8, ViAUInt16, ViAUInt32, ViAUInt64	Data written to the bus (8 bits for viMoveOut8Ex , 16 bits for viMoveOut16Ex , 32 bits for viMoveOut32Ex , and 64 bits for viMOveOut64Ex).

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

See Also

viMoveIn8Ex, viMoveIn16Ex, viMoveIn32Ex, viMoveIn64Ex, viOut8Ex, viOut16Ex, viOut32Ex, viOut64Ex

viOpen

Syntax

viOpen(ViSession sesn, ViRsrc rsrcName, ViAccessMode accessMode, ViUInt32 timeout, ViPSession 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.

Whether **viOpen** actually determines the presence of the device opened depends on the **Address check** property set in Connection Expert for the device. <u>Click</u> <u>here</u> for more information on this interaction.

Parameters

Name	Dir	Туре	Description
sesn	IN	ViSession	Resource Manager session (should always be the Default Resource Manager for VISA returned from <u>viOpenDefaultRM</u>).
rsrcName	IN	ViRsrc	Unique symbolic name (VISA address) of a resource. (See the following tables.) Can also be a VISA alias (defined in the Agilent Connection Expert utility).
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.)
timeout	IN	ViUInt32	If the <i>accessMode</i> parameter requires a lock, this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Otherwise, this parameter is ignored. (Must use VI_NULL in VISA 1.0.) Note: The timeout parameter affects ONLY the LOCK, it does not impact the overall viOpen command timing.
vi	OUT	ViPSession	Unique logical identifier reference to a session.

Address String Grammar for *rsrcName* Parameter

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

PXI	PXI[bus]::device[::function][::INSTR]
PXI	PXI[interface]::bus-device[.function][::INSTR]
PXI	PXI[interface]::CHASSISchassis::SLOTslot[::FUNCfunction][::INSTR]
PXI	PXI[interface]::MEMACC
TCPIP	TCPIP[board]::host address::port::SOCKET
TCPIP	TCPIP[board]::host address[::HiSLIP device name[,HiSLIP port]][::INSTR]
TCPIP	TCPIP[board]::host address[::LAN device name]::INSTR
USB	USB[board]::manufacturer ID::model code::serial number[::USB interface number][::INSTR]
VXI	VXI[board]::MEMACC
VXI	VXI[board]::VXI logical address[::INSTR]
VXI	VXI[board][::VXI logical address]::BACKPLANE

Examples of Address Strings for rsrcName Parameter

Address String	Description
ASRL1::INSTR	A serial device located on port 1.
GPIB::1::0::INSTR	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB2::INTFC	Interface or raw resource for GPIB interface 2.
GPIB-VXI::9::INSTR	A VXI device at logical address 9 in a GPIB-VXI controlled VXI system.
GPIB-VXI1::MEMACC	Board-level register access to GPIB-VXI interface number 1.
"MyDMM"	A device for which the VISA Alias myDMM has been created in the Agilent Connection Expert utility
PXI0::21::INSTR	PXI device 21 on bus 0
PXI0::3-18.2::INSTR	Function 2 on PXI device 18 on bus 3
PXI0::3-18::INSTR	PXI device 18 on bus 3.
PXI0::CHASSIS1::SLOT4::INSTR	PXI device in slot 4 of chassis 1.
PXI0::MEMACC	Access to system controller memory available to devices in the PXI system.
TCPIP0::[fe80::1]::hislip0::INSTR	A TCP/IP device using HiSLIP located at IPv6 IP address fe80::1.
TCPIP::devicename @company.com::INSTR	TCPIP device using VXI-11 located at the specified address. This uses the default LAN Device Name of <i>inst0</i> .
TCPIP0::1.2.3.4::999::SOCKET TCPIP0::	Raw TCPIP access to port 999 at the specified address.
[fe80::218:e77f]::999::SOCKET	Raw TCPIP access to port 999 at the specified IPv6 address.

USB::0x1234::0x5678::A22- 5::INSTR	A USB device with manufacturer ID 0x1234, model code 0x5678, and serial number A22-5. This uses the device's first available USBTMC interface. This is usually number 0.
VXI::1::BACKPLANE	Mainframe resource for chassis 1 on the default VXI system, which is interface 0.
VXI::MEMACC	Board-level register access to the VXI interface.
VXI0::1::INSTR	A VXI device at logical address 1 in VXI interface VXI0.

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	Operation completed successfully.	
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_ALLOC	Insufficient system resources to open a session.	
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.	
VI_ERROR_INV_ACC_MODE	Invalid access mode.	
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.	
VI_ERROR_NSUP_OPER	The given <i>sesn</i> does not support this function. For VISA, this function is supported only by the Default Resource Manager 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_RSRC_NFOUND	Insufficient location information or resource not present in the system.	
VI_ERROR_TMO	A session to the resource could not be obtained within the specified <i>timeout</i> period.	

See Also

viClose Address Check, viFindRsrc, and viOpen

viOpenDefaultRM

Syntax

viOpenDefaultRM(ViPSession 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	<u>Dir</u>	Туре	Description
sesn	OUT	ViPSession	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_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.
VI_ERROR_SYSTEM_ERROR	The VISA system failed to initialize.

See Also

viOpen, viFindRsrc, viClose

viOut8Ex, viOut16Ex, viOut32Ex, and viOut64Ex

Syntax

- viOut8Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViUInt8 buf8);
- viOut16Ex(ViSession vi, ViUInt16 space, ViBusAddress64 offset64, ViUInt16 buf16);
- viOut32Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViUInt32 buf32);
- viOut64Ex(ViSession vi, ViUInt16 space, ViBusAddress64
 offset64, ViUInt64 buf64);

Description

This VISA 4.0 (and later) function writes an 8-bit, 16-bit, 32-bit, or 64-bit word to the specified memory space (assigned memory base + *offset*). This function takes the 8-bit, 16-bit, 32-bit, or 64-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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.		
VI_PXI_CFG_SPACE	Address the PCI configuration space.		
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.		
VI PXI ALLOC SPACE	Access physical locally allocated memory.		

The high-level operations **viOut8Ex**, **viOut16Ex**, **viOut32Ex**, and **viOut64Ex** operate independently from the low-level operations (**viMapAddressEx**, **viPeek8**, **viPeek16**, **viPeek32**, **viPeek64 viPoke8**, **viPoke16**, **viPoke32**, and **viPoke64**). The high-level and low-level operations are independent regardless of the configured state of the hardware that is used to perform memory accesses.

The offset specified in the **viOut8Ex**, **viOut16Ex**, **viOut32Ex**, and **viOut64** operations for an INSTR resource is the offset address relative to the device's allocated address base for the corresponding address space specified. For a MEMACC resource, the offset parameter specifies an absolute address.

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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
space	IN	ViUInt16	Specifies the address space. (See the following table.)
offset64	IN	ViBusAddress64	Offset (in bytes) of the starting address or register to write to.
buf8, buf16, buf32, or buf64	IN	ViUInt8, ViUInt16, ViUInt32, ViUInt64	Data written to bus (8 bits for viOut8 , 16 bits for viOut16 , 32 bits for viOut32 , and 64 bits for viOut64).

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

See Also

viIn8Ex, viIn16Ex, viIn32Ex, viIn64Ex, viPoke8, viPoke16, viPoke32, viPoke64, viMoveOut8Ex, viMoveOut16Ex, viMoveOut32Ex, viMoveOut64Ex

viOut8, viOut16, viOut32, and viOut64

Syntax

- viOut8(ViSession vi, ViUInt16 space, ViBusAddress offs ViUInt8 buf8);
- viOut16(ViSession vi, ViUInt16 space, ViBusAddress offset, ViUInt16 buf16);
- viOut32(ViSession vi, ViUInt16 space, ViBusAddress
 offset, ViUInt32 buf32);
- viOut64(ViSession vi, ViUInt16 space, ViBusAddress
 offset, ViUInt64 buf64); [VISA 4.0 and later]

Description

This function writes an 8-bit, 16-bit, 32-bit, or 64-bit word to the specified memory space (assigned memory base + *offset*). This function takes the 8-bit, 16-bit, 32-bit, or 64-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.

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_A64_SPACE	Address the A64 address space of VXI/MXI bus.		
VI_PXI_CFG_SPACE	Address the PCI configuration space.		
VI_PXI_BAR0_SPACE – VI_PXI_BAR5_SPACE	Address the specified PCI memory or I/O space.		
VI PXI ALLOC SPACE	Access physical locally allocated memory.		

The high-level operations **viOut8**, **viOut16**, **viOut32**, and **viOut64** operate independently from the low-level operations (**viMapAddress**, **viPeek8**, **viPeek16**, **viPeek32**, **viPeek64 viPoke8**, **viPoke16**, **viPoke32**, and **viPoke64**). The high-level and low-level operations are independent regardless of the configured state of the hardware that is used to perform memory accesses.

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

The offset specified in the viOut8, viOut16, viOut32, and viOut64 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, VI_A32_SPACE, or VI_A64_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, A32, or A64 space.

Parameters

Name	Dir	Туре	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.
buf8, buf16, buf32, or buf64	IN	ViUInt8, ViUInt16, ViUInt32, ViUInt64	Data written to bus (8 bits for viOut8 , 16 bits for viOut16 , 32 bits for viOut32 , and 64 bits for viOut64).

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

See Also

viIn8, viIn16, viIn32, viIn64, viPoke8, viPoke16, viPoke32, viPoke64, viMoveOut8, viMoveOut16, viMoveOut32, viMoveOut64

viParseRsrc

Syntax

viParseRsrc(ViSession sesn, ViRsrc rsrcName, VIPUInt16
 intfType, VIPUInt16 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 <u>viFindRsrc</u> and recognized by <u>viOpen</u>. 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**. Agilent VISA recognizes aliases created with the Agilent Connection Expert utility.

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	<u>Dir</u>	Туре	Description
sesn	IN	ViSession	Resource Manager session (should always be the Default Resource Manager for VISA returned from <u>viOpenDefaultRM</u>).
rsrcName	IN	ViRsrc	Unique symbolic name (VISA address or VISA alias) of a resource.
intfType	OUT	VIPUInt16	Interface type of the given resource string.
intfNum	OUT	VIPUInt16	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_ALLOC	Insufficient system resources to parse the string.		
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.		
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.		
VI_ERROR_NSUP_OPER	The given <i>sesn</i> does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.		
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.		

See Also

viFindRsrc, viOpen, viParseRsrcEx

viParseRsrcEx

Syntax

viParseRsrcEx(ViSession sesn, ViRsrc rsrcName, VIPUInt: intfType, VIPUInt16 intfNum, ViString rsrcClass, ViString unaliasedExpandedRsrcName, ViString aliasIfExists)

Description

This function parses a resource string to get extended interface information. 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 VISA address (resource descriptor) would use without actually opening a session to it.

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

The value returned in *unaliasedExpandedRsrcName* are, in most cases, identical to the VISA-defined canonical resource name. However, there may be cases where the canonical name includes information that the driver may not know until the resource has actually been opened. In these cases, the value returned in this parameter must be semantically similar.

The value returned in *aliasIfExists* allows programmatic access to user-defined aliases. If a VISA implementation does not implement aliases, the return value must be an empty string. If a VISA implementation allows multiple aliases for a single resource, then the implementation must pick one alias (in an implementation-defined manner) and return it in this parameter. Agilent VISA recognizes aliases defined in the Agilent Connection Expert utility; if you define multiple aliases for a single resource, *aliasIfExists* will return the first alias found in the Windows registry. (This ordering is not deterministic; you should write your code to expect *aliasIfExists* to return any alias in this situation.)

Parameters

Name	<u>Dir</u>	Туре	Description
sesn	IN	ViSession	Resource Manager session (should always be the Default Resource Manager for VISA returned from <u>viOpenDefaultRM</u>).
rsrcName	IN	ViRsrc	Unique symbolic name of a resource.
intfType	OUT	VIPUInt16	Interface type of the given resource string.
intfNum	OUT	VIPUInt16	Board number of the interface of the given resource string.
rsrcClass	OUT	ViString	Specifies the resource class (for example, "INSTR") of the given resource string, as defined in <u>VISA Resource Classes</u> .
unaliasedExpandedRsrcName	OUT	ViString	This is the expanded version of the given resource string. The format should be similar to the VISA-defined canonical resource name.
aliasIfExists	OUT	ViString	Specifies the user-defined alias for the given resource string, if a VISA implementation allows aliases and an alias exists for the given resource string.

Special Value for *aliasIfExists* **Parameter**

Value	Action Description
VI_NULL	Do not return the alias.
Special Value for unaliasedE	xpandedRsrcName Parameter
VI_NULL	Do not return the full resource name.

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.		
VI_WARN_EXT_FUNC_NIMPL	The operation succeeded, but a lower level driver did not implement the extended functionality.		
Error Codes	Description		
VI_ERROR_ALLOC	Insufficient system resources to parse the string.		
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.		
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.		
VI_ERROR_NSUP_OPER	The given <i>sesn</i> does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session.		
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.		

-

See Also

viFindRsrc, viOpen, viParseRsrc

viPeek8, viPeek16, viPeek32, and viPeek64

Syntax

viPeek8(ViSession vi, ViAddr addr, ViPUInt8 val8);

```
viPeek16(ViSession vi, ViAddr addr, ViPUInt16 val16);
```

```
viPeek32(ViSession vi, ViAddr addr, ViPUInt32 val32);
```

```
viPeek64(ViSession vi, ViAddr addr, ViPUInt64 val64);
[VISA 4.0 and later]
```

Description

This function reads an 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddress</u> call.

Note: *ViAddr* is defined as a void *. To do pointer arithmetic, you must cast this to an appropriate type (**ViUInt8**, **ViUInt16**, **ViUInt32**, or **ViUInt64**). 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.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
addr	IN	ViAddr	Specifies the source address to read the value.
val8, val16, val32, or val64	OUT	ViPUInt8, ViPUInt16, ViPUInt32, ViPUInt64	Data read from bus (8 bits for viPeek8 , 16 bits for viPeek16 , 32 bits for viPeek32 , and 64 bits for viPeek64).

Return Values

None

See Also

<u>viPoke8, viPoke16, viPoke32, viPoke64, viMapAddress, viIn8, viIn16, viIn32</u> and <u>viIn64</u>

viPoke8, viPoke16, viPoke32, and viPoke64

Syntax

viPoke8(ViSession vi, ViAddr addr, ViUInt8 val8); viPoke16(ViSession vi, ViAddr addr, ViUInt16 val16); viPoke32(ViSession vi, ViAddr addr, ViUInt32 val32); viPoke64(ViSession vi, ViAddr addr, ViUInt64 val64); [VISA 4.0 and later]

Description

This function takes an 8-bit, 16-bit, 32-bit, or 64-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 <u>viMapAddress</u> call.

Note: *ViAddr* is defined as a void *. To do pointer arithmetic, you must cast this to an appropriate type (**ViUInt8**, **ViUInt16**, ViUInt32, or ViUInt64). 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.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
addr	IN	ViAddr	Specifies the destination address to store the value.
val8, val16, val32, or val64	IN	ViUInt8, ViUInt16, ViUInt32, or ViUInt64	Data read from bus (8 bits for viPoke8 , 16 bits for viPoke16 , 32 bits for viPoke32 , and 64 bits for viPoke64).

Return Values

None

See Also

viPeek8, viPeek16, viPeek32, viPeek64, viMapAddress, viOut8, viOut16, viOut32, viOut64

viPrintf

Syntax

Description

This formatted IO 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.

Note: The viWrite operation performs the actual low-level I/O to the device. Therefore, you should not use the viWrite and viPrintf operations in the same session.

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

Name	<u>Dir</u>	Туре	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.	

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

For example, *writefmt* could be the string: "The value is '%h5d'". In this example the characters "The value is " are sent to the device unchanged. Then there is the modifier "%h" and the format code" 5d" meaning the argument must be a short or unsigned short (%h) and a five digit integer (5d).

writefmt Special Formatting Characters

The following table lists the possible *writefmt* special characters and 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.
W	Sends a backslash (\) character.

writeFmt Format Specifiers

Format specifiers consist of a % (percent) followed by an optional modifier, followed by a format code. It is in the form:

%[modifier]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.

writefmt Modifiers

Every format specifier starts with the % character and ends with a conversion character (format code). 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.

<u>Modifier</u>	For use with Format Codes	Description
		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 a negative modifier is used, e.g., $-5d$).
An integer specifying field width.	d, f, s	An asterisk (*) may be present in lieu of a field width modifier, in which case an extra <i>arg</i> is used. This <i>arg</i> must be an integer representing the field width. Special case: For the @H, @Q, and @B flags, the <i>field width</i> includes the #H, #!, and #B strings, respectively.
		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

An integer specifying precision.	d, f, s	 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.
h	d, b, B	The h modifier promotes the argument to a short or unsigned short, depending on the format code type.
1	d, f, b, B	The l modifier (lower case letter "L") promotes the argument to a long or unsigned long.
L	f	The L modifier promotes the argument to a long double parameter.
Z	b, B	The \mathbf{z} modifier promotes the argument to an array of floats.
Z	b, B	The ${f Z}$ modifier promotes the argument to an array of doubles.
A comma (,) followed by an integer n , where n represents the array size.	%d, %f	The corresponding argument is interpreted as a reference to the first element of an array of size <i>n</i> . The first <i>n</i> elements of this list are printed in the format specified by the format code. An asterisk (*) may be present after the "," modifier, in which case an extra <i>arg</i> is used. This <i>arg</i> must be an integer representing the array size of the given type.
@1	%d, %f	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (e.g., 123).
@2	%d, %f	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, %f	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).
@Н	%d, %f	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 #HXXX, where <i>XXX</i> is a hexadecimal number (e.g., #HAF35B).</hexadecimal
@Q	%d, %f	Converts to an IEEE 488.2 defined <octal data="" numeric="" response="">. The number is represented in a base of eight form. The number is of the form #Q<i>YYY</i>, where <i>YYY</i> is an octal number (e.g., #Q71234).</octal>
		Converts to an IEEE 488.2 defined <binary numeric<="" td=""></binary>

```
@B %d, %f RESPONSE DATA>. The number is represented in a base two form. The number is of the form #BZZZ.., where ZZZ.. is a binary number (e.g., #B011101001).
```

writeFmt Format Codes

Format Code	For use with Modifier	Description		
%		Send the ASCII percent (%) character.		
С		Argument type: A character to be sent.		
d		Argument type: An integer.		
	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	<i>arg</i> is a short integer.		
	, array size	<i>arg</i> points to an array of integers (or long or short integers, depending on the length modifier) of size <i>array size</i> . The elements of this array are separated by <i>array size</i> -1 commas and output in the specified format.		
f		Argument type: A floating point number		
	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	<i>arg</i> is a double float.		
	Length modifier L	<i>arg</i> is a long double.		
		<i>ara</i> points to an array of floats (or doubles or long doubles)		

arg points to an array of floats (or doubles or long doubles), depending on the length modifier) of size *array size*. The

2	, array size	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.
b		Argument type: A location of a block of data.
fo	Default unctionality	The data block is sent as an IEEE 488.2 <definite length<br="">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 <i>field width</i> or <i>precision</i> modifier is not allowed with this format code.</definite>
	* (asterisk)	An asterisk may be present instead of the count. In such a case, two <i>args</i> are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second <i>arg</i> 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.
	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.
1	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.
В		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 arbitrary="" block="" length="" response<br="">DATA>. This format involves sending an ASCII LF character with the END indicator set after the last byte of the block.</indefinite>
у		Argument type: A location of block binary data.
fr	Default unctionality	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 <i>field width</i> or <i>precision</i> modifier is not allowed with this format code.
	*	A* (asterisk) may be present instead of the count. In such a case, two <i>args</i> are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second <i>arg</i> 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 !ol byte order modifier is present, the data is sent in little endian format. Otherwise, the data is sent in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different.
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.

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

Additional Notes

- Up to four *arg* parameters may be required to satisfy a % format conversion request. In the case where multiple *args* are required, they must 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.

See Also

<u>viVPrintf</u>

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 <u>viPrintf</u> and <u>viScanf</u> 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 <u>viFlush</u>.

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

Parameters

Name	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
writeFmt	IN	ViString	<i>ViString</i> 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_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.		
VI_ERROR_INV_FMT	A format specifier in the <i>writeFmt</i> or <i>readFmt</i> string is invalid.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	Could not perform read/write operation because of I/O error.		
VI_ERROR_NSUP_FMT	The format specifier is not supported for current argument type.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout occurred before read/write operation completed.		

See Also

viPrintf, viScanf, viVQueryf

viRead

Syntax

```
viRead(ViSession vi, ViPBuf buf, ViUInt32 count,
ViPUInt32 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. It is possible to have one, two, or all three of these conditions satisfied at the same time.

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

Note: You must set specific attributes to make the read terminate under specific conditions. See <u>VISA Resource Classes</u> for details.

Note: If you are using **viRead** in Visual Basic 6, see <u>Notes on Using</u> <u>viRead/viWrite in Visual Basic 6</u> for information on modifying its declaration to allow efficient reading and writing of numeric arrays.

Parameters

Name	Dir	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
buf	OUT	ViPBuf	Represents the location of a buffer to receive data from device.
count	IN	ViUInt32	Number of bytes to be read.
retCount	OUT	ViPUInt32	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_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_ASRL_PARITY	A parity error occurred during transfer.		
VI_ERROR_BERR	Bus error occurred during transfer.		
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_INV_SETUP	Unable to start read function because setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_IO	An unknown I/O error occurred during transfer.		
VI_ERROR_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_NSUP_OPER	The given <i>vi</i> does not support this function.		
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.		
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.		
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before function completed.		

See Also

<u>viWrite</u> <u>Notes on Using viRead/viWrite in Visual Basic 6</u>

Notes on Using viRead/viWrite in Visual Basic 6

The visa32.bas file supplied with VISA declares the buffer parameter in **viRead** and **viWrite** as a String data type. If you need to read or write data to a numeric array, it can be very cumbersome and inefficient to convert string data to numeric array data. It is much more efficient to add new declarations for **viRead** and **viWrite** to the visa32.bas file and call through these modified declarations. This allows you to read and write directly to a numeric array without having to convert to or from a string.

Note: Do not modify the original visa32.bas file that was installed with the IO Libraries Suite. Instead, copy visa32.bas to your local project directory and modify the copy. When using the **Project** > **Add Module** command in VB6, be sure you add the local copy of the visa32.bas file to your project and not the visa32.bas file from the original installation directory!

VB6 does not allow overloaded function declarations; this means that if you need to declare additional functions, their names must be unique.

Using **viRead** as an example, the original declaration in visa32.bas for **viRead** is as follows:

Declare Function viRead Lib "VISA32.DLL" Alias "#256"

ByVal vi As Long, ByVal Buffer As String, ByVal count As Long, retCount As Long) As Long

To create versions of **viRead** that can handle a byte and an integer array, you can copy the **viRead** declaration in visa32.bas and insert two copies of it back into the file, but rename these copies and change the buffer data type as shown:

Declare Function viRead Lib "VISA32.DLL" Alias "#256"

(ByVal vi As Long, ByVal Buffer As String, ByVal count As Long, retCount As Long) As Long

Declare Function viReadByte Lib "VISA32.DLL" Alias
 "#256" _

```
(ByVal vi As Long, ByRef Buffer As Byte, ByVal count
As Long, retCount As Long) As Long
Declare Function viReadInteger Lib "VISA32.DLL" Alias
"#256"
(ByVal vi As Long, ByRef Buffer As Integer, ByVal
count As Long, retCount As Long) As Long
```

The modified portions of the new declarations are shown in **bold**. If you need to read additional data types, you can create your own modified function declarations for the data types you need.

Note that in the original declaration, the string was declared as ByVal. This is needed because VB6 will properly marshal a unicode VB6 string to an ASCII string and pass the pointer to the start of the ASCII buffer when the string is passed 'ByVal'. For numeric arrays, however, no marshalling is performed; you need to pass the address of the beginning of the numeric array to VISA. That requires the ByRef keyword.

Once these new declarations have been added, you can call the newly declared functions from your VB6 program as follows:

```
Dim retCount As Long
Dim arraySize As Long
Dim intArray(20) As Integer
' Set arraySize and retCount to the maximum number
' of elements that the array can hold.
' arraySize = (UBound(intArray) - LBound(intArray) + 1) * 2
' Read the array from the test instrument and set
' the retCount variable number of bytes read.
' Remember that retCount needs to be divided
' by the size each element to get the number of
' elements read.
' err = viReadInteger(vi, intArray(0), arraySize, retCount)
```

You can add declarations for **viWrite** variations if you want to use **viWrite** to handle numeric arrays directly.

Remember, though, that the count variables indicate the size in bytes, not in elements. You must adjust the count value based on the size of the numeric element you are reading and writing.

See Also

<u>viRead</u>, <u>viWrite</u> <u>Using the VISA C API in Microsoft Visual Basic 6</u>

viReadAsync

Syntax

viReadAsync(ViSession vi, ViPBuf buf, ViUInt32 count, ViPJobId 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.

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 <u>viTerminate</u>.

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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
buf	OUT	ViPBuf	Represents the location of a buffer to receive data from the device.
count	IN	ViUInt32	Number of bytes to be read.
jobId	OUT	ViPJobId	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_QUEUE_ERROR	Unable to queue read operation.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	
VI_ERROR_IN_PROGRESS	Unable to start a new asynchronous operation while another asynchronous operation is in progress.	

Using viReadAsync

Programming Tip: Performing multiple asynchronous operations

simultaneously: The VISA asynchronous functions viMoveAsync, viReadAsync and viWriteAsync initiate I/O operations to a device on a separate thread which allows the main thread to continue without blocking when doing I/O. VISA allows you to initiate multiple simultaneous asynchronous operations on a single VISA session, but the Agilent IO Libraries Suite allows only a single thread at a time from a given session to access the device. To perform multiple asynchronous operations simultaneously, you can work around this limitation by opening multiple sessions to the device and doing one VISA asynchronous call on each session.

The queuing method is commonly used when an immediate response from your application is not needed (refer to the Agilent VISA User Guide). 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.
- Use viReadAsync to obtain the specific *jobId* from the session you are monitoring.
- When ready to query, use the viWaitOnEvent function with VI_EVENT_IO_COMPLETION to check for queued events.

After 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.

The following example demonstrates this by reading *IDN? as the event.

```
// viReadAsync Example
#include <stdio.h>
#include <stdlib.h>
#include "visa.h"
void main (int argc, char **argv)
£
    printf("viReadAsyncExample\n");
    if ( argc < 2 )
     {
        printf( "A viReadAsync Example\n\n" );
         printf( " usage: %s [VISA address or alias]\n", argv[0] );
         exit(1);
    3
    ViStatus status;
    ViSession drm, vi;
    char *rsrcName = argv[1];
    status = viOpenDefaultRM(&drm);
    printf(" Opening '%s'\n", rsrcName);
    status = viOpen( drm, rsrcName, VI_NULL, VI_NULL, &vi);
    if (status < VI SUCCESS)
     {
        printf("ERROR: Unable to open '%s'\n", rsrcName);
        exit(1);
    }
    status = viEnableEvent(vi, VI EVENT IO COMPLETION, VI QUEUE, VI NULL);
    // Do a *IDN guery to the device
    status = viPrintf(vi, "*IDN?\n");
    char cbuf[256];
    ViJobId jobId;
    status = viReadAsync(vi, (ViPBuf)cbuf, sizeof(cbuf), &jobId);
    ViEventType eventType;
    ViEvent eventContext;
    status = viWaitOnEvent(vi, VI EVENT IO COMPLETION, 2000, &eventType, &eventContext);
    ViEventType returnedEventType;
    ViJobId returnedJobId;
    ViStatus returnedStatus;
    char *returnedBuffer;
    ViUInt32 returnedCount;
    char returnedOperName[256];
    status = viGetAttribute(eventContext, VI_ATTR_EVENT_TYPE, &returnedEventType);
    status = viGetAttribute(eventContext, VI_ATTR_JOB_ID, &returnedJobId);
status = viGetAttribute(eventContext, VI_ATTR_STATUS, &returnedStatus);
status = viGetAttribute(eventContext, VI_ATTR_BUFFER, &returnedBuffer);
    status = viGetAttribute(eventContext, VI ATTR RET COUNT, &returnedCount);
    status = viGetAttribute(eventContext, VI ATTR OPER NAME, returnedOperName);
    status = viClose(eventContext);
    status = viClose(vi);
    status = viClose(drm);
}
```

See Also

viRead, viTerminate, viWrite, viWriteAsync

viReadSTB

Syntax

viReadSTB(ViSession vi, ViPUInt16 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. For a session to a USB instrument, this function sends the READ_STATUS_BYTE command on the control pipe.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to the session.
status	OUT	ViPUInt16	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. Description	
Completion Code		
VI_SUCCESS	Operation completed successfully.	
Error Codes	Description	
VI_ERROR_BERR	Bus error occurred during transfer.	
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).	
VI_ERROR_NCIC	The interface associated with the given <i>vi</i> is not currently the controller in charge.	
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).	
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this function.	
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.	
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> 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.	

viReadToFile

Syntax

viReadToFile (ViSession vi, ViConstString fileName, ViUInt32 count, ViPUInt32 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	<u>Dir</u>	Туре	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	ViPUInt32	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_MAX_CNT	The number of bytes read is equal to <i>count</i> .	
VI_SUCCESS_TERM_CHAR	The specified termination character was read.	
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_ASRL_PARITY	A parity error occurred during transfer.	
VI_ERROR_BERR	Bus error occurred during transfer.	
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.	
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_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_INV_SETUP	Unable to start read function because setup is invalid (due to attributes being set to an inconsistent state).	
VI_ERROR_IO	An unknown I/O error occurred during transfer.	
VI_ERROR_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_NSUP_OPER	The given <i>vi</i> does not support this function.	
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.	
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.	
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

VI_ERROR_TMO

Timeout expired before function completed.

See Also

viRead, viWriteFromFile

viScanf

Syntax

viScanf(ViSession vi, ViString readFmt, arg1, arg2,...

Description

This formatted IO function receives data from a device, formats it according to a format specifier (*readFmt*), and stores the data in the *arg* parameter list. The format specifier can have white-space characters and ordinary characters. The whit-space 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.

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

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
readFmt	IN	ViString	String describing the format for arguments. Refer to the tables below for <i>readFmt modifiers</i> and <i>format codes</i> .
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.

readFmt Format Specifiers

The *readFmt* string consists of a %, optional *modifier* flags, and a *format code*, in that sequence. It is of the form:

%[modifier]format code

where the optional *modifier* describes the data format, and *format code* indicates the data type. One and only one *format code* should be performed at the specifier sequence. A format specifier 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.

readFmt Modifiers

<u>Modifier</u>	For use with Format Codes	Description
An integer (representing the <i>field width)</i>	%s,% c, % []	It specifies the maximum field width that the argument will take. A # may also appear instead of the integer <i>field width</i> , in which case the next <i>arg</i> is a reference to the <i>field width</i> . This <i>arg</i> is a reference to an integer for % c and % s . The <i>field width</i> is not allowed for % d or % f format codes.

h	d, b	The h modifier promotes the argument to be a reference to a short integer or unsigned short integer, depending on the format code.
1	d, f, b	The l (lower case "L") modifier promotes the argument to point to a long integer or unsigned long integer.
L	f	The L modifier promotes the argument to point to a long double floating point parameter.
Z	b	The \mathbf{z} modifier promotes the argument to point to an array of floats.
Z	b	The \mathbf{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.
A comma (" , ") followed by an integer n , where n	%d, %f	The corresponding argument is interpreted as a reference to the first element of an array of size <i>n</i> . The first <i>n</i> elements of this list are printed in the format specified by the conversion character.
represents the array size.		A number sign (#) may be present after the" ," modifier, in which case an extra <i>arg</i> is used. This <i>arg</i> must be an integer representing the array size of the given type.
@1	%d,%f	Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (e.g., 123).
@2	%d,% f	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).
@Н	%d, %f	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 #HXXX, where XXX is a hexadecimal number (e.g., #HAF35B).</hexadecimal
@Q	%d,% f	Converts to an IEEE 488.2 defined <octal data="" numeric="" response="">. The number is represented in a base of eight form. The number is of the form #Q<i>YYY</i>, where <i>YYY</i> is an octal number (e.g., #Q71234).</octal>
@B	%d, %f	Converts to an IEEE 488.2 defined <binary data="" numeric="" response="">. The number is represented in a base two form. The number is of the form #B<i>ZZZ</i>, where <i>ZZZ</i> is a binary number (e.g., #B011101001).</binary>

readFmt Format Codes

	For use	
Format	with	Description
Code	Modifiers	Description
C		Argument type: A reference to a character. White space in the device input stream is not ignored when using c .
	Default functionality	A character is read from the device and stored in the parameter.
	field width	<i>field width</i> number of characters are read and stored at the reference location (the default field width is 1). No NULL character is added at the end of the data block.
d		Argument type: A reference to an integer.
	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<br="">DATA> (#H, #Q, and #B).</non-decimal></decimal
	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	<i>arg</i> is a reference to a short integer. Rounding is performed according to IEEE 488.2 rules (0.5 and up).
	, array size	<i>arg</i> 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.
	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<br="">PROGRAM DATA> (NRf), or <non-decimal NUMERIC PROGRAM DATA> (#H, #Q, and #B).</non-decimal </decimal>
	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 ${f L}$	arg is a reference to a long double number.

	, array size	<i>arg</i> 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.
	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.
		This flag gives the maximum string size. If the <i>field width</i> 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.
	field width	In the case of <i>field width</i> characters already read before encountering a white space, additional characters are read and discarded until a white space character is found.
		In the case of <i>#field width</i> , the actual number of characters that were copied into the user array, not counting the trailing NULL character, read are stored back in the integer pointed to by the first argument.
b		Argument type: A reference to a data array.
		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.</arbitrary
	Default functionality	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 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 ${f Z}$	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
	Default functionality	Characters are read from the device until the first END indicator is received. The character on which the END indicator was received is included in the buffer.
	field width	This flag gives the maximum string size. If an END indicator is not received before <i>field width</i> number of characters, additional characters are read and discarded until an END indicator arrives. <i>#field width</i> has the same meaning as in % s .
Т		Argument type: A reference to a string.
	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 <i>field width</i> number of characters, additional characters are read and discarded until a linefeed character arrives. <i>#field width</i> has the same meaning as in % s .
у		Argument type: A location of block binary data.
		The data block is read as raw binary data. The format specifier sequence should have a flag describing the <i>array size</i> , which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the <i>array size</i> contains a # sign, two arguments are used.
	Default functionality	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 !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 to native computer format.
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.

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

Additional Notes

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

See Also

<u>viVScanf</u>

viSetAttribute

Syntax

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.

The error code VI_ERROR_RSRC_LOCKED is returned only if the specified attribute is read/write and global, and the resource is locked by another session.

Parameters

Name	<u>Dir</u>	Туре	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.
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_ATTR_READONLY	The specified attribute is read-only.	
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.	
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_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

<u>viGetAttribute</u>. See <u>VISA Resource Classes</u> 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 which buffer 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	Dir	Туре	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_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.
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 <i>vi</i> has been locked for this kind of access.

See Also

<u>viFlush</u>

viSPrintf

Syntax

Description

Same as <u>viPrintf</u>, 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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
buf	OUT	ViPBuf	Buffer where data are to be written.
writeFmt	IN	ViString	String describing the format for arguments.
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 Codes	Description	
VI_SUCCESS	Parameters were successfully formatted.	
Error Codes	Description	
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient system resources.	
VI_ERROR_INV_FMT	A format specifier in the <i>writeFmt</i> string is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_NSUP_FMT	A format specifier in the <i>writeFmt</i> string is not supported.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

<u>viPrintf</u>

viSScanf

Syntax

```
viSScanf(ViSession vi, ViBuf buf, ViString readFmt, arg
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 <u>viScanf</u>, except data are read from a user-specified buffer instead of a device.

Parameters

Name	<u>Dir</u>	Туре	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 <i>ViVAList</i> .
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 Codes	Description	
VI_SUCCESS	Data were successfully read and formatted into <i>arg</i> parameter(s).	
Error Codes	Description	
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient system resources.	
VI_ERROR_INV_FMT	A format specifier in the <i>readFmt</i> string is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_NSUP_FMT	A format specifier in the <i>readFmt</i> string is not supported.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

<u>viScanf</u>

viStatusDesc

Syntax

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

<u>Name</u>	<u>Dir</u>	Туре	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);
```

This function requests a VISA 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.

Name	Dir	Туре	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 Codes	Description	
VI_SUCCESS	Request serviced successfully.	
Error Codes	Description	
VI_ERROR_INV_DEGREE	Invalid degree specified.	
VI_ERROR_INV_JOB_ID	Invalid job identifier specified.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	

See Also

viReadAsync, viWriteAsync, viMoveAsync

viUninstallHandler

Syntax

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

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 or which the handler reference and the *userHandle* matches are uninstalled.

Name	Dir	Туре	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_IO_COMPLETIONNotification that an asynchronous
operation has completed.VI_EVENT_PXI_INTRNotification that a vendor-specific PXI
interrupt was received from the device.VI_EVENT_SERVICE_REQNotification that a device is requesting
service.VI_EVENT_TRIGNotification that a hardware trigger was
received from a device.VI_EVENT_VXI_SIGPNotification that a VXI signal or VXI
interrupt has been received from a device.

Special Value for *handler* **Parameter**

Value

Action Description

Uninstall all the handlers with the matching value in the *UserHandle* parameter.

VI_ANY_HNDLR

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 handler successfully uninstalled.
Error Codes	Description
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event.
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_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).

See Also

See the handler prototype <u>viEventHandler</u> for its parameter description.

See the <u>viEnableEvent</u> description for information about enabling different event handling mechanisms. See individual event descriptions for context definitions.

viUnlock

Syntax

viUnlock(ViSession vi);

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

<u>Name</u>	<u>Dir</u>	Туре	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	Session was successfully unlocked.		
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 <i>vi</i> 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 <u>Programming with VISA</u> in the *Agilent VISA User's Guide*.

viUnmapAddress

Syntax

viUnmapAddress(ViSession vi);

This function unmaps memory space previously mapped by the <u>viMapAddress</u> function.

Name	Dir	Туре	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	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 <i>vi</i> 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);

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

This operation unmaps only one trigger mapping per call. If <u>viMapTrigger</u> 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.

Name	Dir	Туре	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 Values for trgSrc Param	leter		
Value			Action Description
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_TTL0 - VI_TRIG_TTL7			Unmap the specified VXI TTL trigger line.
Special Values for trgDest Para	neter		

Value

VI_TRIG_ALL

VI_TRIG_ECL0 -VI_TRIG_ECL1

VI_TRIG_PANEL_IN

VI_TRIG_PANEL_OUT

VI_TRIG_TTL0 -VI_TRIG_TTL7

Action Description

Unmap all trigger lines to which *trigSrc* is currently connected.

Unmap the specified VXI ECL trigger line.

Unmap the controller's front panel trigger input line.

Unmap the controller's front panel trigger output line.

Unmap the specified VXI TTL trigger line.

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

See Also

BACKPLANE resource description

viUsbControlIn

Syntax

viUsbControlIn(ViSession vi, ViInt16 bmRequestType, ViInt16 bRequest, ViUInt16 wValue, ViUInt16 wIndex, ViUInt16 wLength, ViPBuf buf, ViPUInt16 retCnt)

This function can be used to request arbitrary data from a USB device on the default control port. The user must be aware of how to use each parameter based on the relevant USB base or class specification, or based on a vendor-specific request definition.

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
bmRequestType	IN	ViInt16	Bitmap field for defining the USB control port request. The bitmap fields are as defined by the USB specification. The direction bit must be device-to-host.
bRequest	IN	ViInt16	Request ID for this transfer. The meaning of this value depends on <i>bmRequestType</i> .
wValue	IN	ViUInt16	Request value for this transfer.
wIndex	IN	ViUInt16	Specifies the interface or endpoint index number, depending on <i>bmRequestType</i> .
wLength	IN	ViUInt16	Length of the data in bytes to request from the device during the Data stage. If this value is 0, then <i>buf</i> is ignored.
buf	OUT	ViPBuf	Actual data received from the device during the Data stage. If <i>wLength</i> is 0, then this parameter is ignored.
retCnt	OUT	ViPUInt16	Actual number of bytes received from the device during the Data stage.

Special Value for retCnt Parameter

Value

Action Description

VI_NULL

Do not return the actual number of bytes read from the control pipe.

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_MASK	The value in <i>bmRequestType</i> does not have the direction bit set to the correct value.		
VI_ERROR_IO	Could not perform operation because of I/O error.		
VI_ERROR_INV_PARAMETER	The upper 8 bits of <i>bmRequestType</i> or <i>bRequest</i> are not zero.		
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 <i>vi</i> does not support this operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.		

See Also

See the <u>USB INSTR</u> resource description.

viUsbControlOut

Syntax

viUsbControlOut(ViSession vi, ViInt16 bmRequestType, ViInt16 bRequest, ViUInt16 wValue, ViUInt16 wIndex, ViUInt16 wLength, ViBuf buf)

This function can be used to send arbitrary data to a USB device on the default control port. The user must be aware of how to use each parameter based on the relevant USB base or class specification, or based on a vendor-specific request definition.

Since the USBTMC specification does not currently define any standard control port requests in the direction of host-to-device, this function is intended for use with only vendor-defined requests. However, this function implementation should not check the *bmRequestType* parameter for this aspect.

Name	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
bmRequestType	IN	ViInt16	Bitmap field for defining the USB control port request. The bitmap fields are as defined by the USB specification. The direction bit must be host-to-device.
bRequest	IN	ViInt16	Request ID for this transfer. The meaning of this value depends on <i>bmRequestType</i> .
wValue	IN	ViUInt16	Request value for this transfer.
wIndex	IN	ViUInt16	Specifies the interface or endpoint index number, depending on <i>bmRequestType</i> .
wLength	IN	ViUInt16	Length of the data in bytes to send to the device during the Data stage. If this value is 0, then <i>buf</i> is ignored.
buf	IN	ViBuf	Actual data to send to the device during the Data stage. If <i>wLength</i> is 0, then this parameter is ignored.
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_MASK	The value in <i>bmRequestType</i> does not have the direction bit set to the correct value.		
VI_ERROR_IO	Could not perform operation because of I/O error.		
VI_ERROR_INV_PARAMETER	The upper 8 bits of <i>bmRequestType</i> or <i>bRequest</i> are not zero.		
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 <i>vi</i> does not support this operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.		

See Also

<u>USB INSTR</u> 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 <u>viPrintf</u>, except that the *ViVAList* parameters list provides the parameters rather than separate *arg* parameters.

Using viVPrintf in Visual Basic 6

Some of **viPrintf**'s variable arguments are references to primitive (byte, integer, long, float, etc.) types, meaning that the values themselves can be changed by the function. There is no equivalent in VB6 for variable argument lists with reference arguments, so no direct translation is available. Instead, use **viVPrintf** in conjunction with the undocumented VB6 **VarPtr** function to create an array of pointers to arguments than can be passed to the VISA functions. See <u>viVPrintf/viVScanf Example using String</u>for sample VB6 code.

Parameters

Name	Dir	Туре	Description	
vi	IN	ViSession	Unique logical identifier to a session.	
writeFmt	IN	ViString	The format string to apply to parameters in <i>ViVAList</i> . 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 Codes	Description		
VI_SUCCESS	Parameters were successfully formatted.		
Error Codes	Description		
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.		
VI_ERROR_INV_FMT	A format specifier in the <i>writeFmt</i> string is invalid.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	Could not perform write function because of I/O error.		
VI_ERROR_NSUP_FMT	A format specifier in the <i>writeFmt</i> string is not supported.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before write function completed.		

See Also

<u>viPrintf</u> <u>viVPrintf/viVScanf Example using String</u>

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 <u>viQueryf</u>, except that the *ViVAList* parameters list provides the parameters rather than the separate *arg* parameter list in *viQueryf*.

Using viVQueryf in Visual Basic 6

The **viPrintf**, **viScanf**, and **viQueryf** VISA functions take variable argument lists in C. Some of **viScanf**'s and **viQueryf**'s variable arguments are references to primitive (byte, integer, long, float, etc.) types, meaning that the values themselves can be changed by the function. There is no equivalent in VB6 for variable argument lists with reference arguments, so no direct translation is available. Instead, use the 'V' form of these functions (**viVPrintf**, **viVScanf**, and **viVQueryf**), in conjunction with the undocumented VB6 **VarPtr** function to create an array of pointers to arguments than can be passed to the VISA functions. See **viVQueryf** Example with String and Indefinite Length Block for sample code in VB6.

Parameters

Name	<u>Dir</u>	Туре	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 to apply to read parameters in <i>ViVAList</i> .		
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 Codes	Description		
VI_SUCCESS	Successfully completed the query operation.		
Error Codes	Description		
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.		
VI_ERROR_INV_FMT	A format specifier in the <i>writeFmt</i> string or <i>readFmt</i> string is invalid.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	Could not perform read/write function because of I/O error.		
VI_ERROR_NSUP_FMT	The format specifier is not supported for current argument type.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before read/write operation completed.		

See Also

viVPrintf, viVScanf, viQueryf

viVQueryf Example with String and Indefinite Length Block

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 <u>viScanf</u>, except that the *ViVAList* parameters list provides the parameters rather than separate *arg* parameters.

Using viVScanf in Visual Basic 6

The **viPrintf**, **viScanf**, and **viQueryf** VISA functions take variable argument lists in C. Some of **viScanf**'s and **viQueryf**'s variable arguments are references to primitive (byte, integer, long, float, etc.) types, meaning that the values themselves can be changed by the function. There is no equivalent in VB6 for variable argument lists with reference arguments, so no direct translation is available. Instead, use the 'V' form of these functions (**viVPrintf**, **viVScanf**, and **viVQueryf**), in conjunction with the undocumented VB6 **VarPtr** function to create an array of pointers to arguments than can be passed to the VISA functions.

The examples below are presented as self-contained VB6 Sub's with comments explaining the various features. You can modify and adapt the code in the examples to your specific situation.

- viVPrintf/viVScanf Example using String
- <u>viVScanf</u> Example Returning a Double Array
- <u>viVScanf</u> Example Reading an IEEE 488 Definite Length Block and Returning a Byte Array
- <u>viVScanf</u> Example Reading an IEEE 488 Indefinite Length Block and Returning a Byte Array

Parameters

Name	<u>Dir</u>	Туре	Description	
vi	IN	ViSession	Unique logical identifier to a session.	
readFmt	IN	ViString	The format string to apply to parameters in <i>ViVAList</i> . 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 Codes	Description		
VI_SUCCESS	Data were successfully read and formatted into <i>arg</i> parameter(s).		
Error Codes	Description		
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.		
VI_ERROR_INV_FMT	A format specifier in the <i>readFmt</i> string is invalid.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	Could not perform read function because of I/O error.		
VI_ERROR_NSUP_FMT	A format specifier in the <i>readFmt</i> string is not supported.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before read function completed.		

See Also

<u>viScanf</u>

viVPrintf/viVScanf Example using String

viVScanf Example Returning a Double Array

viVScanf Example Reading an IEEE 488 Definite Length Block and Returning a Byte Array

viVScanf Example Reading an IEEE 488 Indefinite Length Block and Returning a Byte Array

viVSPrintf

Syntax

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

Description

Same as <u>viVPrintf</u>, 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	<u>Dir</u>	Туре	Description	
vi	IN	ViSession	Unique logical identifier to a session.	
buf	OUT	ViPBuf	Buffer where data are to be written.	
writeFmt	IN	ViString	The format string to apply to parameters in <i>ViVAList</i> .	
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 Codes	Description		
VI_SUCCESS	Parameters were successfully formatted.		
Error Codes	Description		
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.		
VI_ERROR_INV_FMT	A format specifier in the <i>writeFmt</i> string is invalid.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	Could not perform read function because of I/O error.		
VI_ERROR_NSUP_FMT	A format specifier in the <i>writeFmt</i> string is not supported.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		

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 <u>viVScanf</u>, except data are read from a user-specified buffer rather than a device.

Parameters

Name	<u>Dir</u>	Туре	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 <i>ViVAList</i> .	
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 Codes	Description	
VI_SUCCESS	Data were successfully read and formatted into <i>arg</i> parameter(s).	
Error Codes	Description	
VI_ERROR_ALLOC	The system could not allocate a formatted I/O buffer because of insufficient resources.	
VI_ERROR_INV_FMT	A format specifier in the <i>readFmt</i> string is invalid.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_NSUP_FMT	A format specifier in the <i>readFmt</i> string is not supported.	
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.	

See Also

viSScanf, viVScanf

viVxiCommandQuery

Syntax

```
viVxiCommandQuery(ViSession vi, ViUInt16 mode, ViUInt3:
    cmd, ViPUInt32 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	<u>Dir</u>	Туре	Description
vi	IN	ViSession	Unique logical identifier to a session.
mode	IN	ViUInt16	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	ViPUInt32	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_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_RESP16*	Get 16-bit response from previous query.	
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 Codes	Description		
VI_SUCCESS	Operation completed successfully.		
Error Codes	Description		
VI_ERROR_BERR	Bus error occurred during transfer.		
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.		
VI_ERROR_INV_MODE	The value specified by the <i>mode</i> parameter is invalid.		
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 <i>vi</i> does not support this function.		
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.		
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.		
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.		
VI_ERROR_RESP_PENDING	A previous response is still pending, causing a multiple query error.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before function completed.		

See Also

INSTR Resource Description

viWaitOnEvent

Syntax

viWaitOnEvent(ViSession vi, ViEventType inEventType, ViUInt32 timeout, ViPEventType outEventType, ViPEven 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. **Important:** C/C++ programs do this, see <u>Note on Context Return in visa32.vb</u> below.

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 <u>viClose</u> 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 <u>viReadSTB</u> function to read the status byte of the service request.

Event Name	Attributes	Data Type	Range	
		<u></u>		
VI_EVENT_SERVICE_REQ	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_F	
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE VI_ATTR_SIGP_STATUS_ID	ViEventType ViUInt16	VI_EVENT_VXI_STOP 0 to FFFF _h	
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIG VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1	
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_COMPL N/A N/A 0 to FFFFFFFh N/A	
VI_EVENT_VXI_VME_INTR Not supported by Agilent VISA	VI_ATTR_EVENT_TYPE VI_ATTR_INTR_STATUS_ID VI_ATTR_RECV_INTR_LEVEL	ViEventType ViUInt32 ViInt16	VI_EVENT_VXI_VME_ 0 to FFFFFFFh 1 to 7, VI_UNKNOWN_LEVEJ	
VI_EVENT_PXI_INTR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_PXI_INTR	
VI_EVENT_USB_INTR	VI_ATTR_EVENT_TYPE VI_ATTR_USB_RECV_INTR_SIZE VI_ATTR_USB_RECV_INTR_DATA VI_ATTR_STATUS	ViEventType ViUInt16 ViBuf ViStatus	VI_EVENT_USB_INTR 0 to FFFFh N/A N/A	

Instrument Control (INSTR) Resource Events

Memory Access (MEMACC) Resource Events

Event Name	Attributes	Data Type	Range
	VI_ATTR_EVENT_TYPE	ViEventType ViStatus	VI_EVENT_IO_COMPLETION
	VI_ATTR_STATUS	ViStatus ViIobId	N/A N/A
VI_EVENT_IO_COMPLETION	VI_ATTR_BUFFER	ViBuf	N/A
	ATTR_RET_COUNT	ViUInt32	0 to FFFFFFFFh

GPIB Bus Interface (INTFC) Resource Events

vent Name Attributes Da Ty		Data Type	Range	
VI_EVENT_GPIB_CIC	VI_ATTR_EVENT_TYPE VI_ATTR_GPIB_RECV_CIC_STATE	ViEventType ViBoolean	VI_EVENT_GPIB_CIC VI_TRUE VI_FALSE	
VI_EVENT_GPIB_TALK	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_TALŀ	
VI_EVENT_GPIB_LISTEN	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_GPIB_LIST	
VI_EVENT_CLEAR	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_CLEAR	
VI_EVENT_TRIGGER	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIGGER VI_TRIG_SW	
VI_EVENT_IO_COMPLETION	VI_ATTR_EVENT_TYPE VI_ATTR_STATUS VI_ATTR_JOB_ID VI_ATTR_BUFFER VI_ATTR_RET_COUNT VI_ATTR_RET_COUNT_32 VI_ATTR_RET_COUNT_64 VI_ATTR_OPER_NAME	ViEventType ViStatus ViJobId ViBuf ViUInt32 ViString	VI_EVENT_IO_COMPL N/A N/A N/A 0 to FFFFFFFh N/A	

VXI Mainframe Backplane (BACKPLANE) Resource Events

Event Name	Attributes	Data Type	Range
VI_EVENT_TRIG	VI_ATTR_EVENT_TYPE VI_ATTR_RECV_TRIG_ID	ViEventType ViInt16	VI_EVENT_TRIG VI_TRIG_TTL0 to VI_TRIG_ VI_TRIG_ECL0 to VI_TRIG_
VI_EVENT_VXI_VME_SYSFAIL	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSF
VI_EVENT_VXI_VME_SYSRESET	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_VXI_VME_SYSF

TCPIP Socket (SOCKET) Resource

Event Name	Attributes	Data Type	Range	
	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_IO_COMPLETION	
	VI_ATTR_STATUS	ViStatus	N/A	
VI	_EVENT_	_IO_	COMPLETION	V
----	---------	------	------------	---
----	---------	------	------------	---

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	Dir	Туре	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.	
outEventType	OUT	ViPEventType	Logical identifier of the event actually received.	
outContext	OUT	ViPEvent	A handle specifying the unique occurrence of an event.	
Special Values for <i>outEventType</i> Parameter				
Value			Description	
VI_NULL			Do not return the type of event.	
Special Value f	or <i>outCo</i>	ntext 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 <i>inEventType</i> type available for this session.		
Error Codes	Description		
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_TMO	Specified event did not occur within the specified time period.		

Note on Context Return in visa32.cs and visa32.vb

It is not possible to pass a 'null' pointer for the 'ref context' parameter in viWaitOnEvent as it is defined in visa32.cs and visa32.vb. This means that VISA will not close the context and when viWaitOnEvent is called in a loop, the system will eventually run out of resources.

The solution is to close the returned context after a successful return from viWaitOnEvent. Here is some C# sample code that works:

```
...
int context = 0;
err = visa32.viWaitOnEvent(vi, visa32.VI_EVENT_SERVICE_REQ, 500,
ref eventType, ref context);
if (err >= visa32.VI_SUCCESS)
{
    visa32.viClose(context);
}
...
```

See Also

See <u>Programming with VISA</u> in the *Agilent VISA User's Guide* for more information on event handling.

viWrite

Syntax

```
viWrite(ViSession vi, ViBuf buf, ViUInt32 count,
ViPUInt32 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.

Note: If you are using **viWrite** in Visual Basic 6, see <u>Notes on Using</u> <u>viRead/viWrite in Visual Basic 6</u> for information on modifying its declaration to allow efficient reading and writing of numeric arrays.

Parameters

Name	<u>Dir</u>	Туре	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	ViPUInt32	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	Transfer completed.		
Error Codes	Description		
VI_ERROR_BERR	Bus error occurred during transfer.		
VI_ERROR_CONN_LOST	The I/O connection for the given session has been lost.		
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.		
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_INV_SETUP	Unable to start write function because setup is invalid (due to attributes being set to an inconsistent state).		
VI_ERROR_IO	Unknown I/O 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 de-asserted).		
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this function.		
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.		
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before function completed.		

See Also

<u>viRead</u> <u>Notes on Using viRead/viWrite in Visual Basic 6</u>

viWriteAsync

Syntax

viWriteAsync(ViSession vi, ViBuf buf, ViUInt32 count, ViPJobId jobId);

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 <u>viTerminate</u> to abort the operation, or with an I/O Completion event to identify which asynchronous write operation completed.

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.

Programming Tip: Performing multiple asynchronous operations

simultaneously: The VISA asynchronous functions viMoveAsync, viReadAsync and viWriteAsync initiate I/O operations to a device on a separate thread which allows the main thread to continue without blocking when doing I/O. VISA allows you to initiate multiple simultaneous asynchronous operations on a single VISA session, but the Agilent IO Libraries Suite allows only a single thread at a time from a given session to access the device. To perform multiple asynchronous operations simultaneously, you can work around this limitation by opening multiple sessions to the device and doing one VISA asynchronous call on each session.

Parameters

Name	<u>Dir</u>	Туре	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.
jobId	OUT	ViPJobId	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_QUEUE_ERROR	Unable to queue write operation.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_IN_PROGRESS	Unable to start a new asynchronous operation while another asynchronous operation is in progress.		

See Also

viRead, viTerminate, viWrite, viReadAsync

viWriteFromFile

Syntax

viWriteFromFile (ViSession vi, ViConstString fileName
ViUInt32 count, ViPUInt32 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	<u>Dir</u>	Туре	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	Specifies number of bytes to be written.	
retCount	OUT	ViPUInt32	Number of bytes actually transferred.	
Special Value for <i>retCount</i> 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	Transfer completed.		
Error Codes	Description		
VI_ERROR_BERR	Bus error occurred during transfer.		
VI_ERROR_CONN_LOST	I/O connection for a session has been lost.		
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_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).		
VI_ERROR_IO	An unknown I/O error occurred during transfer.		
VI_ERROR_NCIC	The interface associated with the given <i>vi</i> is not currently the controller in charge.		
VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).		
VI_ERROR_NSUP_OPER	The given <i>vi</i> does not support this function.		
VI_ERROR_OUTP_PROT_VIOL	Device reported input protocol error during transfer.		
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.		
VI_ERROR_RAW_RW_PROT_VIOL	Violation of raw write protocol occurred during transfer.		
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by <i>vi</i> has been locked for this kind of access.		
VI_ERROR_TMO	Timeout expired before function completed.		

See Also

viWrite, viReadToFile

VISA Attributes

This topic summarizes the attributes of the VISA Template, VISA resource classes, and shows applicable interface types for each resource class supported by Agilent VISA.

Note: Although the Servant Device-Side (SERVANT) Resource is defined by the *VXIplug&play Systems Alliance VISA Library* specification, the SERVANT Resource is not supported in Agilent VISA and is not described in this help. The SERVANT Resource is intended for advanced users who need to write firmware that exports device functionality across multiple resources.

VISA Template Attributes

See the topic below for a list of attributes for the VISA Template.

VISA Template Attributes

Resource Class Descriptions

See these topics for five resource classes supported by Agilent VISA. (the SERVANT resource cClass is not supported by Agilent VISA). The description for each resource class includes a resource overview, resource attributes, resource events, and resource operations (functions).

Instrument Control (INSTR) Resource

Memory Access (MEMACC) Resource

<u>GPIB</u> Bus Interface (INTFC) Resource

VXI Mainframe Backplane (BACKPLANE) Resource

TCPIP Socket (SOCKET) Resource

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 Attributes 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.

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, USB, VXI	Device operations (reading, writing, trigg	
GPIB Bus Interface (INTFC)	Generic, GPIB	Raw GPIB interface operations (reading, triggering, etc.).	
Memory Access (MEMACC)	Generic, GPIB-VXI, VXI	Address space of a memory-mapped bus VXIbus.	
VXI Mainframe Backplane (BACKPLANE)	Generic, GPIB-VXI, VXI (GPIB-VXI Backplane not supported)	VXI-defined operations and properties of (or chassis) in a VXIbus system.	
Servant Device-Side Resource (SERVANT)	Not Supported (GPIB, VXI, TCPIP)	Operations and properties of the capabilit and a device's view of the system in which it exists.	
TCPIP Socket (SOCKET)	Generic, TCPIP	Operations and properties of a raw netwo 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)				
ТСРІР	Instrument Control (INSTR) TCPIP Socket (SOCKET)				
USB	Instrument Control (INSTR)				
VXI	Instrument Control (INSTR) Memory Access (MEMACC) VXI Mainframe Backplane (BACKPLANE)				

See Also:

VISA Attribute Values

Agilent-Defined VISA Attributes

This topic summarizes the attributes specific to Agilent's VISA implementation. They are defined in the visa.h file distributed with Agilent VISA.

Note: The #define AGVISA_ATTRIBUTES statement must appear before the #include <visa.h> statement if you are using any Agilent-defined attributes.

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).

SCPI/TULIP Information

Attribute	Acco Priv	ess ileges	Data Type	Range	Used by	Description
VI_AGATTR_LOCKWAIT	RW	Local	ViBoolean	VI_TRUE / VI_FALSE	all VISA sessions	Indicates whether a lock-abiding VISA function that has been exclusively locked by another session will: • wait the timeout value to acquire the lock and then return VI_ERROR_TMO if it can't (VI_TRUE case) or • return a VI_ERROR_RSRC_LOCKED error immediately (VI_FALSE case).

TCPIP Attributes

Attribute	Access Privileges		Data Type	Range	Used by	
VI_AGATTR_INTERFACE_PROTOCOL	RO	Local	ViUInt32	VI_AGPROT_VXI11 VI_AGPROT_SICLLAN VI_AGPROT_HISLIP VI_AGPROT_UNKNOWN (on non-LAN sessions)	INSTR, INTFC	
VI_AGATTR_REMOTE_INTF_TYPE	RO	Local	ViUInt16	VI_INTF_GPIB VI_INTF_ASRL VI_INTF_VXI VI_AGINTF_USRDEF VI_AGINTF_LANINST VI_AGINTF_RSIB VI_AGINTF_SOCKET VI_AGINTF_HISLIP VI_INTF_USB	INSTR, INTFC	

Export Attributes

Attribute	Acc Priv	ess ⁄ileges	Data Type	Range	Used by	D
VI_AGATTR_EXPORT_ENABLED	RW	Global	ViBoolean	VI_TRUE / VI_FALSE	All VISA sessions	D(sh vi: in Tł de
VI_AGATTR_FIND_ONLY_EXPORTED_RSRCS	RW	Local	ViBoolean	VI_TRUE / VI_FALSE	DefaultRM sessions only	W on V atti rei vi T t wl be thi V atti

GPIB Attributes

Attribute	Acc Priv	ess ⁄ileges	Data Type	Range	Used by	1
VI_AGATTR_GPIB_T1_DELAY	RW	Global	ViiNT32	VI_AG_GPIB_T1DELAY_MIN to VI_AG_GPIB_T1DELAY_MAX (value is in nanoseconds)	GPIB INTFC resources	T ti iv iv C S S S C S S C t t t C d t t t t v t t v t

Miscellaneous Attributes

Attribute	Acco Priv	ess ileges	Data Type	Range	Used by	Description
VI_AGATTR_ALLOW_LOCAL_SPACE_FIFC	RW	Local	ViBoolean	VI_TRUE / VI_FALSE (default)	All VISA sessions	This attribute was added in 3.0. The VISA 3.0 spec requination that in viMove, the srcIncrement/destIncrement from/to local memory is ign This means FIFO moves fro local memory are disallowed Agilent's VISA implementariallowed FIFO moves from/t memory in VISA implementariallowed FIFO moves from/t memory in VISA implementariallowed FIFO moves from/t memory in VISA 3.0 so this represents a change in behavior to VISA 3.0 Setting this attribute to VI will revert to the pre-VISA behavior and allow FIFO minom/to local memory. Note default behavior can be chan by modifying the registry DWORD value: DefaultAllowLocalSpaceFifin: HKLM\SOFTWARE\Agilen Libraries\CurrentVersioin\V
VI_AGATTR_INTFC_SERIALNUMBER	RO	Global	ViString	N/A	INTFC sessions only	This attribute was added in 15.1. It returns the serial num of an interface. If the interfa does not have a serial numb returns an empty string. Inte that don't support this attrib will return the VI_ERROR_NSUP_ATTR code.
VI_AGATTR_VISA_FRAMEWORK_DIR	RO	Global	ViString	N/A	INTFC sessions only	
VI_AGATTR_IOLIBRARIES_DIR	RO	Global	ViString	N/A	INTFC sessions only	
VI_AGATTR_IOLIBRARIES_REG_PATH	RO	Global	ViString	N/A	INTFC sessions only	

VISA Template Attributes

This topic summarizes the interface that each VISA implementation must incorporate.

VISA Template Attributes

Symbolic Name	Access Privilege		Data Type	Range	Descr
VI_ATTR_RSRC_IMPL_VERSION	RO	Global	ViVersion	0 _h to FFFFFFFh	Resource version t uniquely identifie of the di revisions impleme of a reso
VI_ATTR_RSRC_LOCK_STATE	RO	Global	ViAccessMode	VI_NO_LOCK VI_EXCLUSIVE_LOCK VI_SHARED_LOCK	The curr locking s the reso be unloc locked w exclusiv or locket shared lo
VI_ATTR_RSRC_MANF_ID	RO	Global	ViUInt16	0 _h to 3FFF _h	A value correspo the VXI manufac of the manufac that crea impleme
VI_ATTR_RSRC_MANF_NAME	RO	Global	ViString	N/A	A string correspo the VXI manufac name of manufac that crea impleme
VI_ATTR_RSRC_NAME	RO	Global	ViRsrc	N/A	The unic identifie resource
VI_ATTR_RSRC_SPEC_VERSION***	RO	Global	ViVersion	00500000 _h	Resource version t uniquely identifieversion o VISA specifica which th impleme is compl

VI_ATTR_RM_SESSION	RO	Local	ViSession	N/A	Specifies session c Resource Manager was usec open this session.
VI_ATTR_MAX_QUEUE_LENGTH	R/W*	Local	ViUInt32	1 _h to FFFFFFF _h	Specifies maximu number events th be queue time on t given se
VI_ATTR_RSRC_CLASS	RO	Global	ViString	N/A	Specifies resource (for exar "INSTR
VI_ATTR_USER_DATA	R/W	Local	ViAddr	**	Data use privately applicati particula session. data is n by VISA purposes providec applicati its own t
VI_ATTR_USER_DATA_32	R/W	Local	ViUInt32	0 _h to FFFFFFF _h	Data use privately applicati particula session. data is n by VISA purposes providec applicati its own u
VI_ATTR_USER_DATA_64****	R/W	Local	ViUInt64	0 _h to FFFFFFFFFFFFFFFFFF	Data use privately applicati particula session. data is n by VISA purposes providec applicati its own u

* This attribute becomes RO once **viEnableEvent** has been called for the first time.

** Specified in the relevant VPP-4.3.x framework document.
*** The value of this attribute is a fixed value that reflects the version of the VISA specification to which the implementation is compliant. This value will change with subsequent versions of the specification.

**** Defined only for frameworks that are 64-bit native.

VISA Template Operations

viClose(vi) viGetAttribute(vi, attribute, attrState) viSetAttribute(vi, attribute, attrState) viStatusDesc(vi, status, desc) viTerminate(vi, degree, jobId) viLock(vi, lockType, timeout, requestedKey, accessKey) viUnlock(vi) viEnableEvent(vi, eventType, mechanism, context) viDisableEvent(vi, eventType, mechanism) viDiscardEvents(vi, eventType, mechanism) viDiscardEvents(vi, eventType, timeout, outEventType, outContext) viInstallHandler(vi, eventType, handler, userHandle) viUninstallHandler(vi, eventType, handler, userHandle)

GPIB Bus Interface (INTFC) Resource

This topic 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 <u>viSetAttribute</u>. 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 Privileges Data Type Range

Generic INTFC Resource Attributes

VI_ATTR_DEV_STATUS_BYTE	RW	Global	ViUInt8	0 to FF _h
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_FILE_APPEND_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_INTF_INST_NAME	RO	Global	ViString	N/A
VI_ATTR_INTF_NUM	RO	Global	ViUInt16	0 to FFFF _h
VI_ATTR_INTF_TYPE	RO	Global	ViUInt16	VI_INTF_GPIB
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_DISABLE
VI_ATTR_SEND_END_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_TERMCHAR	RW	Local	ViUInt8	0 to FF _h
VI_ATTR_TERMCHAR_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_TMO_VALUE	RW	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to FFFFFFE _h
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL
VI_ATTR_RD_BUF_SIZE	RO	Local	ViUInt32	N/A
VI_ATTR_WR_BUF_SIZE	RO	Local	ViUInt32	N/A

GPIB-Specific INTFC Resource Attributes

VI_ATTR_GPIB_ADDR_STATE	RO	Global	ViInt16	VI_GPIB_UNADDRESSED VI_GPIB_TALKER VI_GPIB_LISTENER
VI_ATTR_GPIB_HS488_CBL_LEN	RW	Global	ViInt16	1 to 15 VI_GPIB_HS488_DISABLEI VI_GPIB_HS488_NIMPL
VI_ATTR_GPIB_SECONDARY_ADDR	RW	Global	ViUInt16	0 to 30 VI_NO_SEC_ADDR

VI_STATE_ASSERTED

VI_ATTR_GPIB_ATN_STATE	RO	Global	ViInt16	VI_STATE_UNASSERTED VI_STATE_UNKNOWN
VI_ATTR_GPIB_CIC_ STATE	RO	Global	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_GPIB_NDAC_ STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN
VI_ATTR_GPIB_PRIMARY_ADDR	RW	Global	ViUInt16	0 to 30
VI_ATTR_GPIB_REN_STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN
VI_ATTR_GPIB_SRQ_STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN
VI_ATTR_GPIB_SYS_CNTRL_STATE	RW	Global	ViBoolean	VI_TRUE VI_FALSE

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 ope terminates.
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a term 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 devi respond. A timeout value of VI_TMO_INFINITE disables the timeout mecha
VI_ATTR_DEV_STATUS_BYTE	This attribute specifies the 488-style status byte of the local controller associa this session. If this attribute is written and bit 6 (0x40) is set, this device or co 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 VI_FLUSH_WHEN_FULL (default), the buffer is flushed when an END indi written to the buffer, or when the buffer fills up. If the operational mode is set VI_FLUSH_ON_ACCESS, the write buffer is flushed under the same conditi also every time a <u>viPrintf</u> 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 h 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 mo VI_FLUSH_DISABLE (default), the buffer is flushed only on explicit calls to the operational mode is set to VI_FLUSH_ON_ACCESS, the buffer is flushed time a <u>viScanf</u> operation completes.
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether <u>viReadToFile</u> will overwrite (truncate) or app opening a file.
VI_ATTR_RD_BUF_SIZE	This attribute specifies the size of the formatted I/O read buffer. The user can value by calling viSetBuf().
VI_ATTR_WR_BUF_SIZE	This attribute specifies the size of the formatted I/O write buffer. The user can this value by calling viSetBuf().
GPIB-Specific INTFC Resource Attribute	25

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) into

VI_ATTR_GPIB_ATN_STATE	This attribute shows the current state of the GPIB ATN (ATtentioN) interface
VI_ATTR_GPIB_NDAC_STATE	This attribute shows the current state of the GPIB NDAC (Not Data ACcepted) interface line.
VI_ATTR_GPIB_SRQ_STATE	This attribute shows the current state of the GPIB SRQ (Service ReQuest) inte
VI_ATTR_GPIB_CIC_STATE	This attribute shows whether the specified GPIB interface is currently CIC (cc charge).
VI_ATTR_GPIB_SYS_CNTRL_STATE	This attribute shows whether the specified GPIB interface is currently the syst controller. In some implementations, this attribute may be modified only throu 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 s GPIB interface. If HS488 is not implemented, querying this attribute should revalue VI_GPIB_HS488_NIMPL. On these systems, trying to set this attribute return error VI_ERROR_NSUP_ATTR_STATE.
VI_ATTR_GPIB_ADDR_STATE	This attribute shows whether the specified GPIB interface is currently address 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	<u>AP</u>	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	<u>AP</u>	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	<u>AP</u>	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
			Туре	

Unique logical

VI_ATTR_EVENT_TYPE	identifier of the	RO	ViEventType	VI_EVENT_CLEAR
	event.			

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

Event Attribute	Description	<u>AP</u>	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	<u>AP</u>	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	ViBus Size	*
VI_ATTR_RET_COUNT_32	Actual number of elements that were	RO	ViUInt32	0 to FFFFFFFFh

	asynchronously transferred.			
VI_ATTR_RET_COUNT_64**	Actual number of elements that were asynchronously transferred.	RO	ViUInt64	0 to FFFFFFFF FFFFFFh
VI_ATTR_OPER_NAME	The name of the operation generating the event.	RO	ViString	N/A

*The data type is defined in the appropriate VPP 4.3.x framework specification.

**Defined only for operating systems that are 64-bit native.

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)

Instrument Control (INSTR) Resource

This topic 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 <u>viSetAttribute</u>, 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

ASRL Specific INSTR Resource Attributes

Attribute Name	Acc Priv	ess vilege Type	e Range	Default
VI_ATTR_ASRL_AVAIL_NUM	RO	Global ViUIn	t32 0 to FFFFFFFh	0
VI_ATTR_ASRL_BAUD	RW	Global ViUIn	t32 0 to FFFFFFFh	9600
VI_ATTR_ASRL_CTS_STATE	RO	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_DATA_BITS	RW	Global ViUIn	t16 5 to 8	8
VI_ATTR_ASRL_DCD_STATE	RO	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_DSR_STATE	RO	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_DTR_STATE	RW	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_END_IN	RW	Local ViUIn	t16 VI_ASRL_END_NONE VI_ASRL_END_LAST_BIT VI_ASRL_END_TERMCHAR	VI_ASRL_E
VI_ATTR_ASRL_END_OUT	RW	Local ViUIn	t16 VI_ASRL_END_NONE VI_ASRL_END_LAST_BIT VI_ASRL_END_TERMCHAR VI_ASRL_END_BREAK	VI_ASRL_E
VI_ATTR_ASRL_FLOW_CNTRL	RW	Global ViUIn	t16 VI_ASRL_FLOW_NONE VI_ASRL_FLOW_XON_XOFF VI_ASRL_FLOW_RTS_CTS VI_ASRL_FLOW_DTR_DSR	VI_ASRL_F
VI_ATTR_ASRL_PARITY	RW	Global ViUIn	t16 VI_ASRL_PAR_NONE VI_ASRL_PAR_ODD VI_ASRL_PAR_EVEN VI_ASRL_PAR_MARK VI_ASRL_PAR_SPACE	VI_ASRL_P.
VI_ATTR_ASRL_REPLACE_CHA	R RW	Local ViUIn	t8 0 to FFh	0
VI_ATTR_ASRL_RI_STATE	RO	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_RTS_STATE	RW	Global ViUIn	t16 VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_ASRL_STOP_BITS	RW	Global ViUIn	t16 VI_ASRL_STOP_ONEVI_ASRL_STO	P_TWO VI_ASRL_S
VI_ATTR_ASRL_XOFF_CHAR	RW	Local ViUIn	t8 0 to FFh	<ctrl+s>(13)</ctrl+s>
VI_ATTR_ASRL_XON_CHAR	RW	Local ViUIn	t8 0 to FFh	<ctrl+q>(11</ctrl+q>

Generic INSTR Resource Attributes

Attribute Name	Acce	SS	Data	Range	Default
	Privi	lege	Туре		
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_INTF_INST_NAME	RO	Global	ViString	N/A	N/A
VI_ATTR_INTF_NUM	RO	Global	ViUInt16	0 to FFFFh	0
VI_ATTR_INTF_TYPE	RO	Global	ViUInt16	VI_INTF_VXI VI_INTF_GPIB VI_INTF_GPIB_VXI VI_INTF_ASRL VI_INTF_PXI VI_INTF_TCPIP VI_INTF_USB	N/A
VI_ATTR_TMO_VALUE	R/W	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to FFFFFFEh VI_TMO_INFINITE	2000 msec
VI_ATTR_TRIG_ID	R/W*	Local	ViInt16	VI_TRIG_SW VI_TRIG_TTL0 to VI_TRIG_TTL7 VI_TRIG_ECL0 to VI_TRIG_ECL1	VI_TRIG_SW

*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).

GPIB and GPIB-VXI Specific INSTR Resource Attributes

Symbolic Name	Acc Priv	ess ⁄ilege	Data Type	Range	<u>Default</u>
VI_ATTR_GPIB_PRIMARY_ADDR	RO	Global	ViUInt16	0 to 30	N/A
VI_ATTR_GPIB_READDR_EN	R/W	Local	ViBoolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_GPIB_REN_STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A
VI_ATTR_GPIB_SECONDARY_ADDR	RO	Global	ViUInt16	0 to 31 VI_NO_SEC_ADDR	N/A

VI_ATTR_GPIB_UNADDR_EN

VI_FALSE

GPIB-VXI Specific INSTR Resource Attribute

Attribute Name	Acc	ess	Data Range	Default
	Priv	v ilege	Туре	
VI ATTR INTF PARENT NUM	RO	Global	ViUInt160 to FFFFh	VI ATTR INTF PARI

HiSLIP Specific INSTR Resource Attributes

Attribute Name	Acce Priv	ess ilege	Data Type	Range	Default
VI_ATTR_TCPIP_HISLIP_OVERLAP_EN	R/W	Local	ViBoolean	VI_TRUE, VI_FALSE	Preference returned by device
VI_ATTR_TCPIP_HISLIP_VERSION	RO	Local	ViVersion	N/A	N/A
VI_ATTR_TCPIP_HISLIP_MAX_MESSAGE_KB	R/W	Local	ViUInt32	0 h -0 FFFFFFF h	1024
Massage Daced INSTD Descurree A	ttribu	toc			

Message-Based INSTR Resource Attributes

Attribute Name	Acces Privi	ss lege	Data Type	Range	Default
VI_ATTR_FILE_APPEND_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_False
VI_ATTR_IO_PROT	R/W	Local	ViUInt16	VI_PROT_NORMAL VI_PROT_FDC VI_PROT_HS488 VI_PROT_4882_STRS VI_PROT_USBTMC_VENDOR	VI_ATTR_IO_PF
VI_ATTR_RD_BUF_OPER_MODE	R/W	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_DISABLE	VI_FLUSH_DIS
VI_ATTR_RD_BUF_SIZE	RO	Local	ViUInt32	N/A	N/A
VI_ATTR_SEND_END_EN	R/W	Local	ViBoolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_SUPPRESS_END_EN	R/W	Local	ViBoolean	VI_TRUE VI_FALSE	VI_TRUE
VI_ATTR_TERMCHAR	R/W	Local	ViUInt8	0 to FFh	0ah (newline)
VI_ATTR_TERMCHAR_EN	R/W	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE

VI_ATTR_WR_BUF_OPER_MODE R/W Local ViUInt16 VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL

VI_FLUSH_WH

VI_ATTR_WR_BUF_SIZE RO Local ViUInt32 N/A N/A PXI Specific INSTR Resource Attributes

	Access		Data	
Attribute Name	Privileg	ge	Туре	Range
VI_ATTR_PXI_ACTUAL_LWIDTH	RO	Global	ViInt16	1, 4, 8
VI_ATTR_PXI_BUS_NUM	RO	Global	ViUInt16	0 to 255
VI_ATTR_PXI_CHASSIS	RO	Global	ViInt16	0 to 255 VI_UNKNOWN_CHASSIS
VI_ATTR_PXI_DEV_NUM	RO	Global	ViUInt16	0 to 31
VI_ATTR_PXI_DSTAR_BUS	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_TRIG
VI_ATTR_PXI_DSTAR_SET	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_TRIG
VI_ATTR_PXI_FUNC_NUM	RO	Global	ViUInt16	0 to 7
VI_ATTR_PXI_IS_EXPRESS	RO	Global	ViBoolean	VI_TRUE, VI_FALSE
VI_ATTR_PXI_MAX_LWIDTH	RO	Global	ViInt16	1, 4, 8
VI_ATTR_PXI_MEM_BASE_BAR <i>n</i> (where <i>n</i> is 0,1,2,3,4,5)	RO	Global	ViBusAddress	N/A
VI_ATTR_PXI_MEM_SIZE_BAR <i>n</i> (where <i>n</i> is 0,1,2,3,4,5)	RO	Global	ViBusSize	N/A
VI_ATTR_PXI_MEM_TYPE_BAR <i>n</i> (where <i>n</i> is 0,1,2,3,4,5)	RO	Global	ViUInt16	VI_PXI_ADDR_MEM, VI_PXI_ADDR_IO, VI_PXI_ADDR_NONE
VI_ATTR_PXI_SLOT_LBUS_LEFT	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_SLOT
VI_ATTR_PXI_SLOT_LBUS_RIGHT	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_SLOT
VI_ATTR_PXI_SLOT_LWIDTH	RO	Global	ViInt16	1, 4, 8
VI_ATTR_PXI_SLOTPATH	RO	Global	ViString	N/A
VI_ATTR_PXI_STAR_TRIG_BUS	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_TRIG
VI_ATTR_PXI_STAR_TRIG_LINE	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_TRIG
VI_ATTR_PXI_TRIG_BUS	RO	Global	ViInt16	0 to 32767 VI_UNKNOWN_TRIG

TCPIP Specific INSTR Resource Attributes

Attribute Name	Access	Data Type	Range	Default
	Privilege	_		

VI_ATTR_TCPIP_ADDR	RW	Global	ViString	N/A	N/A
VI_ATTR_TCPIP_DEVICE_NAME	RW	Global	ViString	N/A	N/A
VI_ATTR_TCPIP_HOSTNAME	RW	Global	ViString	N/A	N/A
VI_ATTR_TCPIP_IS_HISLIP	RO	Global	ViBoolean	VI_TRUE, VI_FALSE	N/A

USB Specific INSTR Resource Attributes

Attribute Name	Acce	ess	Data Type	Range	Default
	Priv	ilege	_		
VI_ATTR_USB_INTFC_NUM	RO	Global	ViInt16	0 to 254	0
VI_ATTR_USB_MAX_INTR_SIZE	RW	Local	ViUInt16	0 to FFFFh	N/A
VI_ATTR_USB_PROTOCOL	RO	Global	ViInt16	0 to 255	N/A
vI_ATTR_USB_SERIAL_NUM VXI and GPIB-VXI Speci	^{RO} fic II	Global NSTR Re	ViString Source Attri	N/A butes	N/A

Attribute Name	Acce	ess	Data Type	Range	Default
	Priv	ilege	_		
VI_ATTR_CMDR_LA	RO	Global	ViInt16	0 to 255; VI_UNKNOWN_LA	N/A
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_DEST_BYTE_ORDER	RW	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_ENDIAI
VI_ATTR_FDC_CHNL	RW	Local	ViUInt16	0 to 7	N/A
VI_ATTR_FDC_GEN_SIGNAL_EI	N RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_FDC_MODE	RW	Local	ViUInt16	VI_FDC_NORMAL VI_FDC_STREAM	VI_FDC_NORM
VI_ATTR_FDC_USE_PAIR	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_IMMEDIATE_SERV	RO	Global	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_MAINFRAME_LA	RO	Global	ViInt16	0 to 255; VI_UNKNOWN_LA	N/A
VI_ATTR_MEM_BASE_32	RO	Global	ViBusAddress	N/A	N/A

VI_ATTR_MEM_BASE_64	RO	Global	ViBusAddress6	4 N/A	N/A
VI_ATTR_MEM_SIZE_32	RO	Global	ViBusSize	N/A	N/A
VI_ATTR_MEM_SIZE_64	RO	Global	ViBusSize64	N/A	N/A
VI_ATTR_MEM_SPACE	RO	Global	ViUInt16	VI_A16_SPACE VI_A24_SPACE VI_A32_SPACE VI_A64_SPACE	VI_A16_SPACE
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_SRC_BYTE_ORDER	RW	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_ENDIA
VI_ATTR_VXI_DEV_CLASS	RO	Global	ViUInt16	VI_VXI_CLASS_MEMORY VI_VXI_CLASS_EXTENDEI VI_VXI_CLASS_MESSAGE VI_VXI_CLASS_REGISTER VI_VXI_CLASS_OTHER	N/A)
VI_ATTR_VXI_LA	RO	Global	ViInt16	0 to 511	N/A
VI_ATTR_VXI_TRIG_SUPPORT	RO	Global	ViUInt32	N/A	N/A
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_WIN_BYTE_ORDER	RW*	Local	ViUInt16	VI_BIG_ENDIAN VI LITTLE ENDIAN	VI_BIG_ENDIAI

*For VISA 2.2, the attributes VI_ATTR_WIN_BYTE_ORDER and VI_ATTR_WIN_ACCESS_PRIV are R/W (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).

VXI and GPIB-VXI and PXI Specific INSTR Resource Attributes

Attribute Name	Acce	ess	Data Type	Range	Default
	Priv	ilege	_		
VI_ATTR_DEST_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_SLOT	RO	Global	ViInt16	0 to 10; VI_UNKNOWN_SLO	N/A DT
VI_ATTR_SRC_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_WIN_ACCESS	RO	Local	ViUInt16	VI_NMAPPED VI_USE_OPERS	VI_NMAPPEI

				VI_DEREF_ADDR	
VI_ATTR_WIN_BASE_ADDR_32	RO	Local	ViBusAddress	N/A	N/A
VI_ATTR_WIN_BASE_ADDR_64	RO	Local	ViBusAddress6	4 N/A	N/A
VI_ATTR_WIN_SIZE_32	RO	Local	ViBusSize	N/A	N/A
VI_ATTR_WIN_SIZE_64	RO	Local	ViBusSize64	N/A	N/A
VXI and GPIB-VXI and USB	Specif	ic INS	TR Resour	ce Attributes	

Attribute Name	Access	Data Type	Range	Default
	Privilege	_		

VI_ATTR_4882_COMPLIANT RO Global ViBoolean VI_TRUE, VI_FALSE N/A VXI and GPIB-VXI and USB and PXI Specific INSTR Resource Attributes

Attribute Name	Acces	SS	Data Type	Range	Default
	<u>Privi</u>	lege	-		
	DO				
VI_AITR_MANF_ID	RO	Global	ViUInt16	0 to FFFFh	N/A
VI_ATTR_MANF_NAME	RO	Global	ViString	N/A	N/A
VI_ATTR_MODEL_CODE	RO	Global	ViUInt16	0 to FFFFh	0
VI_ATTR_MODEL_NAME	RO	Global	ViString	N/A	N/A

INSTR Resource Attribute Descriptions

Attribute Name	Description
ASRL Specific INSTR Resource Attributes	
VI_ATTR_ASRL_AVAIL_NUM	This attribute shows the number c
VI_ATTR_ASRL_BAUD	This is the baud rate of the interfa integer so that any baud rate can l used rate such as 300, 1200, 2400
VI_ATTR_ASRL_CTS_STATE	This attribute shows the current s
VI_ATTR_ASRL_DATA_BITS	This is the number of data bits co bits for each frame are located in memory.
VI_ATTR_ASRL_DCD_STATE	This attribute shows the current si signal. The DCD signal is often u carrier (remote modem) on the tel "Receive Line Signal Detect (RLS

VI_ATTR_ASRL_DSR_STATE VI_ATTR_ASRL_DTR_STATE

VI_ATTR_ASRL_END_IN

VI_ATTR_ASRL_END_OUT

VI_ATTR_ASRL_FLOW_CNTRL

VI_ATTR_ASRL_PARITY

VI_ATTR_ASRL_REPLACE_CHAR

VI_ATTR_ASRL_RI_STATE

This attribute shows the current st

This attribute is used to manually (DTR) output signal.

This attribute indicates the metho to VI_ASRL_END_NONE, the rt data is received (or an error occur

If it is set to VI_ASRL_END_TE the character in VI_ATTR_TERN VI_ASRL_END_LAST_BIT, the arrives with its last bit set. For exi to 8, then the read will terminate

This attribute indicates the metho to VI_ASRL_END_NONE, the w written. If it is set to VI_ASRL_E after all the characters for the wri

If it is set to VI_ASRL_END_LA character with the last bit clear, th set. For example, if VI_ATTR_A clear the 8th bit for all but the last the 8th bit set. If it is set to VI_AS the character in VI_ATTR_TERN

If this attribute is set to VI_ATTR mechanism does not use flow con connection are assumed to be larg

If this attribute is set to VI_ATTR mechanism uses the XON and XC transfer mechanism controls inpu buffer is nearly full, and it control when XOFF is received.

If this attribute is set to VI_ATTR mechanism uses the RTS output s control. The transfer mechanism (signal when the receive buffer is I suspending the transmission wher

If this attribute is set to VI_ASRI uses the DTR output signal and the The transfer mechanism controls the receive buffer is nearly full, and transmission when the DSR signal

This attribute can specify multiple multiple values together. Howeve by all serial ports and/or operating <u>Combinations</u> for details.

This is the parity used with every VI_ASRL_PAR_MARK means tl VI_ASRL_PAR_SPACE means tl

This attribute specifies the charac that arrive with errors (such as pa

This attribute shows the current st RI signal is often used by modem

VI_ATTR_ASRL_RTS_STATE	This attribute is used to manually output signal. When the VI_ATTI VI_ASRL_FLOW_RTS_CTS, th read to determine whether the bac unasserting the signal.
VI_ATTR_ASRL_STOP_BITS	This is the number of stop bits us VI_ASRL_STOP_ONE5 indicate
VI_ATTR_ASRL_XOFF_CHAR	This attribute specifies the value of flow control (both directions). If ζ handshaking) is not being used, the flow of the specific spec
VI_ATTR_ASRL_XON_CHAR	This attribute specifies the value of flow control (both directions). If \hat{L} handshaking) is not being used, the flow of the specific s
Generic INSTR Resource Attributes	
VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/ or Programmed I/O (VI_FALSE). have global effects even though it this affects performance and not f
VI_ATTR_INTF_INST_NAME	Human-readable text describing t
VI_ATTR_INTF_NUM	Board number for the given interf
VI_ATTR_INTF_TYPE	Interface type of the given session
VI_ATTR_TMO_VALUE	Minimum timeout value to use, ir VI_TMO_IMMEDIATE means tl to respond. A timeout value of VI mechanism.
VI_ATTR_TRIG_ID	Identifier for the current triggerin
GPIB and GPIB-VXI Specific INSTR Resource Attributes	
VI_ATTR_GPIB_PRIMARY_ADDR	Primary address of the GPIB devi
VI_ATTR_GPIB_READDR_EN	This attribute specifies whether to write operation.
VI_ATTR_GPIB_REN_STATE	This attribute returns the current $\boldsymbol{\varepsilon}$
VI_ATTR_GPIB_SECONDARY_ADDR	Secondary address of the GPIB d
VI_ATTR_GPIB_UNADDR_EN	This attribute specifies whether to each read or write operation.
GPIB-VXI Specific INSTR Resource Attribute	
VI_ATTR_INTF_PARENT_NUM	Board number of the GPIB board
HiSLIP Specific INSTR Resource Attributes	
VI_ATTR_TCPIP_HISLIP_OVERLAP_EN	This enables HiSLIP overlap moc by the instrument on HiSLIP com mode to allow overlapped respon- synchronous mode to detect and r VISA will do a Device Clear oper

VI_ATTR_TCPIP_HISLIP_VERSION

This is the HiSLIP protocol versic example, HiSLIP version 1.0 retu

VI_ATTR_TCPIP_HISLIP_MAX_MESSAGE_KB	This is the maximum HiSLIP mes system in units of kilobytes (1024 message size).
Message-Based INSTR Resource Attributes	
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether \underline{v} when opening a file.
VI_ATTR_IO_PROT	Specifies which protocol to use. I between normal word serial or fas
	In GPIB, you can choose betweer transfers. In ASRL systems, you o transfers, in which case the <u>viAss</u> 488.2-defined strings.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode is set to VI_FLUSH_DISABLE (calls to <u>viFlush</u> . If the operational buffer is flushed every time a <u>viS</u>
VI_ATTR_RD_BUF_SIZE	This attribute specifies the size of modify this value by calling <u>viSe</u>
VI_ATTR_SEND_END_EN	Whether to assert END during the
VI_ATTR_SUPPRESS_END_EN	Whether to suppress the END ind VI_TRUE, the END indicator doo is set to VI_FALSE, the END ind
VI_ATTR_TERMCHAR	Termination character. When the VI_ATTR_TERMCHAR_EN is e operation terminates.
VI_ATTR_TERMCHAR_EN	Flag that determines whether the termination character is received.
VI_ATTR_WR_BUF_OPER_MODE	Determines the operational mode is set to VI_FLUSH_WHEN_FU END indicator is written to the bu
	If the operational mode is set to V flushed under the same conditions completes.
VI_ATTR_WR_BUF_SIZE	This attribute specifies the size of modify this value by calling <u>viSe</u>
PXI Specific INSTR Resource Attributes	
VI_ATTR_PXI_ACTUAL_LWIDTH	Specifies the negotiated link widt
VI_ATTR_PXI_BUS_NUM	PCI bus number of the
VI_ATTR_PXI_CHASSIS	Chassis number in which this dev
VI_ATTR_PXI_DEV_NUM	PCI device number of this device
VI_ATTR_PXI_DSTAR_BUS	Number of the DSTAR bus conne
VI_ATTR_PXI_DSTAR_SET	Specifies the set of PXI_DSTAR
VI_ATTR_PXI_FUNC_NUM	PCI function number of the device devices will also support other fur
VI_ATTR_PXI_IS_EXPRESS	Specifies whether this device is P

VI_ATTR_PXI_MAX_LWIDTH VI_ATTR_PXI_MEM_BASE_BAR*n* VI_ATTR_PXI_MEM_SIZE_BAR*n* VI_ATTR_PXI_MEM_TYPE_BAR*n*

VI_ATTR_PXI_SLOT_LBUS_LEFT VI_ATTR_PXI_SLOT_LBUS_RIGHT

VI_ATTR_PXI_SLOT_LWIDTH VI_ATTR_PXI_SLOTPATH

VI_ATTR_PXI_STAR_TRIG_BUS VI_ATTR_PXI_STAR_TRIG_LINE VI_ATTR_PXI_TRIG_BUS

TCPIP-Specific INSTR Resource Attributes VI_ATTR_TCPIP_ADDR

VI_ATTR_TCPIP_DEVICE_NAME

VI_ATTR_TCPIP_HOSTNAME

VI_ATTR_TCPIP_IS_HISLIP

USB Specific INSTR Resource Attributes VI_ATTR_USB_INTFC_NUM

VI_ATTR_USB_MAX_INTR_SIZE

VI_ATTR_USB_PROTOCOL VI_ATTR_USB_SERIAL_NUM

VXI and GPIB-VXI Specific INSTR Resource Attributes VI_ATTR_CMDR_LA

VI_ATTR_DEST_ACCESS_ PRIV VI_ATTR_DEST_BYTE_ORDER Specifies the maximum link widtl Memory base address assigned to Size of the memory assigned to th Memory type (memory mapped o specified BAR. Slot number or special feature co Slot number or special feature coi device. Specifies the link width used by t Slot path of this device. PXI slot PCI device number and function 1 bridge that routes the module to tl attribute value is device1[.functio Number of the star trigger bus co PXI_STAR line connected to this Number of the trigger bus connec This is the TCPIP address of the c string is formatted in dot-notation This specifies the LAN device na connection. This specifies the host name of th attribute returns an empty string. This specifies whether this resour

Specifies the USB interface numb connected.

Specifies the maximum number o interrupt IN pipe. The default val the interrupt IN pipe.

Specifies the USB protocol numb

This string attribute is the serial n this attribute should be used for d decisions.

Logical address of the commande

This attribute specifies the addres operations, such as *viOutXX* and *v*

This attribute specifies the byte of such as *viOutXX* and *viMoveOutX*

VI_ATTR_FDC_CHNL VI_ATTR_FDC_GEN_SIGNAL_EN

VI_ATTR_FDC_MODE

VI_ATTR_FDC_USE_PAIR

VI_ATTR_IMMEDIATE_SERV

VI_ATTR_MAINFRAME_LA

VI_ATTR_MEM_BASE_32 VI_ATTR_MEM_BASE_64

VI_ATTR_MEM_SIZE_32 VI_ATTR_MEM_SIZE_64

VI_ATTR_MEM_SPACE

VI_ATTR_SRC_ACCESS_PRIV

VI_ATTR_SRC_BYTE_ORDER

VI_ATTR_VXI_DEV_CLASS

VI_ATTR_VXI_LA

VI_ATTR_VXI_TRIG_SUPPORT

VI_ATTR_WIN_ACCESS_PRIV

VI_ATTR_WIN_BYTE_ORDER

This attribute determines which F

Setting this attribute to VI_TRUE the FDC channel is passed back to commander from having to poll tl transfer.

This attribute determines which F mode).

If set to VI_TRUE, a channel pair only one channel will be used.

Specifies whether the given devic running VISA.

This is the logical address of a giv with the lowest logical address. C of the Slot 0 controller or of the p same value. The purpose of this a mainframe. A VISA manufacture consistent across mainframes. If t returned is VI_UNKNOWN_LA.

Base address of the device in VX is applicable to A24 or A32 addre

Size of memory requested by the

VXIbus address space used by the A16/A24, A16/A32, or or A16/A

This attribute specifies the addres operations, such as *viInXX* and *vi*.

This attribute specifies the byte of such as *viInXX* and *viMoveInXX*,

This attribute represents the VXIthe resource belongs:

message based (VI_V

register based (VI_VX

extended (VI_VXI_CL/

memory (VI_VXI_CLA:

VME devices are usually either re (VI_VXI_CLASS_OTHER)

Logical address of the VXI or VN VME device, the logical address i 511.

This attribute shows which VXI t is a bit vector with bits 0-9 corres VI_TRIG_ECL1.

This attribute specifies the addres operations, such as *viMapAddress* mapped window.

This attribute specifies the byte of such as *viMapAddress*, *viPeekXX*

window.

VXI and GPIB-VXI and PXI Specific INSTR Resource Attributes

VI_ATTR_DEST_INCREMENT	This is used in the viMoveOutXX offset is to be incremented after e is 1 (that is, the destination addres and the viMoveOutXX operation
	If this attribute is set to 0, the viN same element, essentially treating
VI_ATTR_SLOT	Physical slot location of the VXIt VI_UNKNOWN_SLOT is return
VI_ATTR_SRC_INCREMENT	This is used in the <i>viMoveInXX</i> o _l is to be incremented after every tr (that is, the source address will be <i>viMoveInXX</i> operation moves from to 0, the <i>viMoveInXX</i> operation w essentially treating the source as a
VI_ATTR_WIN_ACCESS	Modes in which the current wind through operations <i>viPeekXX</i> and by directly dereferencing the addı
VI_ATTR_WIN_BASE_ADDR_32 VI_ATTR_WIN_BASE_ADDR_64	Base address of the interface bus
VI_ATTR_WIN_SIZE_32 VI_ATTR_WIN_SIZE_64	Size of the region mapped to this
VXI, GPIB-VXI, and USB Specific INSTR Resource Attributes	
VI_ATTR_4882_COMPLIANT VXI and GPIB-VXI and USB and PXI Specific INSTR Resource	Specifies whether the device is 48 Attributes
VI_ATTR_MANF_ID	Manufacturer identification numb Subsystem Vendor ID are defined Subsystem Vendor ID. If Subsyst defined for the device, this attribu
VI_ATTR_MANF_NAME	This string attribute is the manufa should be used for display purpos the value can be different betwee
VI_ATTR_MODEL_CODE	Model code for the device. For P2 are defined for the device, then th Subsystem ID and Subsystem Ver attribute value is the PCI Device
VI_ATTR_MODEL_NAME	This string attribute is the model should be used for display purpos the value can be different between

INSTR Resource Events

This resource defines the following events for communication with applications, where AP = Access Privilege.

VI_EVENT_SERVICE_REQ - 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_SEF

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

Event Attribute	Description	<u>AP</u>	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_SERVICE_REQ
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 FFFFh

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

Event Attribute	Description	AP	Data	Range
			Туре	
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 Attribute	Description	<u>AP</u>	Data Type	Range
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous	RO	ViBuf	N/A

	operation.			
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed	RO	ViStatus	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViBus Size	*
VI_ATTR_RET_COUNT_32	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to FFFFFFFFh
VI_ATTR_RET_COUNT_64**	Actual number of elements that were asynchronously transferred.	RO	ViUInt64	0 to FFFFFFFF FFFFFFFFh

*The data type is defined in the appropriate VPP 4.3.x framework specification.

**Defined only for operating systems that are 64-bit native.

VI_EVENT_PXI_INTR - *Notification that a vendor-specific PXI interrupt was received from the device.*

Event Attribute	Description	<u>AP</u>	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_PXI_INTR

VI_EVENT_VXI_VME_INTR - Notification that a VXIbus interrupt was received from the device. NOT IMPLEMENTED IN AGILENT VISA.

Event Attribute	Description	AP	Data	Range
			Туре	

VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_INTR
VI_ATTR_RECV_INTR_LEVEL	VXI interrupt level on which the interrupt was received.	RO	VIInt16	1 to 7, VI_UNKNOWN_LEVEL
VI_ATTR_INTR_STATUS_ID	32-bit status/ID retrieved during the IACK cycle.	RO	ViUInt32	0 to FFFFFFFFh

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

			Data	
Event Attribute	Description	AP	Туре	Raı
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EV
VI_ATTR_USB_RECV_INTR_SIZE	Specifies the size of the data that was received from the USB interrupt-IN pipe. This value will never be larger than the sessions value of VI_ATTR_USB_MAX_INTR_SIZE.	RO	ViUInt16	0 to FF
VI_ATTR_USB_RECV_INTR_DATA	Specifies the actual data that was received from the USB interrupt-IN pipe. Querying this attribute copies the contents of the data to the users buffer. The users buffer must be sufficiently large enough to hold all of the data.	I RO	ViBuf	N/A
VI_ATTR_STATUS	Specifies the status of the read operation from the USB interrupt-IN pipe. If the device sent more data than the user specified in VI_ATTR_USB_MAX_INTR_SIZE, then this attribute value will contain an error code.	RO	ViStatus	N/A

INSTR Resource Operations

viAssertTrigger (vi, protocol) viBufRead (vi, buf, count, retCount) viBufWrite (vi, buf, count, retCount) viClear (vi) viFlush (vi, mask) viGpibControlREN (vi, mode) **viIn8** (vi, space, offset, val8) *viIn16* (vi, space, offset, val16) *viIn32* (vi, space, offset, val32) *viIn64* (vi, space, offset, val64)

viMapAddress (vi, mapSpace, mapBase, mapSize, access, suggested, address)
viMapAddressEx (vi, mapSpace, mapBase64, mapSize, access, suggested, address)
viMemAlloc (vi, size, offset)
viMemAllocEx (vi, size, offset64)
viMemFree (vi, offset)
viMemFreeEx (vi, offset64)

viMove (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length) *viMoveEx* (vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length)

viMoveAsync (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveAsyncEx (vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId)

viMoveIn8 (vi, space, offset, length, buf8) viMoveIn16 (vi, space, offset, length, buf16) viMoveIn32 (vi, space, offset, length, buf32) viMoveIn64 (vi, space, offset, length, buf64)

viMoveIn8Ex(vi, space, offset64, length, buf8) viMoveIn16Ex(vi, space, offset64, length, buf16) viMoveIn32Ex(vi, space, offset64, length, buf32) viMoveIn64Ex(vi, space, offset64, length, buf64)

viMoveOut8 (vi, space, offset, length, buf8) viMoveOut16 (vi, space, offset, length, buf16) viMoveOut32 (vi, space, offset, length, buf32) viMoveOut64 (vi, space, offset, length, buf64)

viMoveOut8Ex(vi, space, offset64, length, buf8)

viMoveOut16Ex(vi, space, offset64, length, buf16) viMoveOut32Ex(vi, space, offset64, length, buf32) viMoveOut64Ex(vi, space, offset64, length, buf64)

viOut8 (vi, space, offset, val8) viOut16 (vi, space, offset, val16) viOut32 (vi, space, offset, val32) viOut64 (vi, space, offset, val64)

viPeek8 (vi, addr, val8) viPeek16 (vi, addr, val16) viPeek32 (vi, addr, val32) viPeek64 (vi, addr, val64) viPoke8 (vi, addr, val8) viPoke16 (vi, addr, val16) viPoke32 (vi, addr, val32) viPoke64 (vi, addr, val64)

```
viPrintf (vi, writeFmt, 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)
viUsbControlIn(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf,
retCnt)
viUsbControlOut(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf)

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)

TCPIP Socket (SOCKET) Resource

This topic 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 <u>viSetAttribute</u>, 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 usually 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 IEEE-488.2-style strings, an attribute setting also allows sending software triggers, querying an IEEE-488-style status byte, and sending a device clear message.

SOCKET Resource Attributes

Note: AP = Access Privileges

Attribute Name	AP		Data —Type	Range	Default
Generic SOCKET Resource Attribu	ites				
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_FILE_APPEND_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	VI_FALSE
VI_ATTR_INTF_INST_NAME	RO	Global	ViString	N/A	N/A
VI_ATTR_INTF_NUM	RO	Global	ViUInt16	0 to FFFF _h	0
VI_ATTR_INTF_TYPE	RO	Global	ViUInt16	VI_INTF_TCPIP	VI_INTF_TCPII
VI_ATTR_IO_PROT	RW	Local	ViUInt16	VI_NORMAL VI_PROT_4882_STRS	VI_NORMAL
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_DISABLE	VI_FLUSH_DIS
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_WR_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL >	VI_FLUSH_WH
VI_ATTR_RD_BUF_SIZE	RO	Local	ViUInt32	N/A	N/A
VI_ATTR_WR_BUF_SIZE	RO	Local	ViUInt32	N/A	N/A
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

RW Local

ViBoolean VI_TRUE VI_FALSE VI_FALSE
SOCKET Resource Attribute Descriptions

Attribute Name	Description
----------------	-------------

Generic SOCKET Resource Attributes

VI_ATTR_DMA_ALLOW_EN	This attribute specifies whether I/O accesses should use DMA (VI_TRUE) or Pro I/O (VI_FALSE). In some implementations, this attribute may have global effects though it is documented to be a local attribute. Since this affects performance and I functionality, that behavior is acceptable.
VI_ATTR_FILE_APPEND_EN	This attribute specifies whether $\underline{viReadToFile}$ will overwrite (truncate) or append v opening a file.
VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_IO_PROT	Specifies which protocol to use.
VI_ATTR_RD_BUF_OPER_MODE	Determines the operational mode of the read buffer. When the operational mode is VI_FLUSH_DISABLE (default), the buffer is flushed only on explicit calls to $viFl$
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. Note: the termination character must be enabled in the program when read SOCKET device that does not support EOI; otherwise the SOCKET read will times the source of the termination character as the source of the source of the termination character as the source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of the termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabled in the program when read source of termination character must be enabl
VI_ATTR_TERMCHAR_EN	Flag that determines whether the read operation should terminate when a termination is received.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of VI_TMO_IMI means that operations should never wait for the device to respond. A timeout value 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 VI_FLUSH_WHEN_FULL (default), the buffer is flushed when an END indicator 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, a every time a <u>viPrintf</u> operation completes.
VI_ATTR_RD_BUF_SIZE	This attribute specifies the size of the formatted I/O read buffer. The user can mo value by calling viSetBuf().
VI_ATTR_WR_BUF_SIZE	This attribute specifies the size of the formatted I/O write buffer. The user can movalue by calling viSetBuf().
TCPIP Specific SOCKET Resource A	ttributes
VI_ATTR_TCPIP_ADDR	This is the TCPIP address of the device to which the session is connected. This stri formatted in dot notation.
VI_ATTR_TCPIP_HOSTNAME	Specifies the host name of the device. If no host name is available, this attribute retempty string.
VI_ATTR_TCPIP_PORT	Specifies the port number for a given TCPIP address. For a TCPIP SOCKET resou 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 synchronous writes get flushed immediately.
VI_ATTR_TCPIP_KEEPALIVE	An application can request that a TCPIP provider enable the use of "keep-alive" pa TCP connections by turning on this attribute. If a connection is dropped as a result alives," the error code VI_ERROR_CONN_LOST is returned to current and subsecalls 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 Attribute	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_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViBus Size	*
VI_ATTR_RET_COUNT_32	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to FFFFFFFF _h
VI_ATTR_RET_COUNT_64**	Actual number of elements that were asynchronously transferred.	RO	ViUInt64	0 to FFFFFFF FFFFFFFh

*The data type is defined in the appropriate VPP 4.3.x framework specification.

******Defined only for operating systems that are 64-bit native.

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)

VXI Mainframe Backplane (BACKPLANE) Resource

This topic 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 <u>viSetAttribute</u>, 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 mainframes in a VXI system as one entity. In these configurations, separate BACKPLANE resources are not possible.

BACKPLANE Resource Attributes

Note: AP = Access Privileges

Attribute Name	AP		Туре	Range	Def
Generic BACKPLANE Resource Attributes					
VI_ATTR_INTF_INST_NAME	RO	Global	ViString	N/A	N/A
VI_ATTR_INTF_NUM	RO	Global	ViUInt16	0 to FFFFh	0
VI_ATTR_INTF_TYPE	RO	Global	ViUInt16	VI_INTF_VXI VI_INTF_GPIB_VXI	N/A
VI_ATTR_TMO_VALUE	RW	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to FFFFFFFEh VI_TMO_INFINITE	2000
VXI and GPIB-VXI Specific BACKPLANE	Resou	irce Attr	ibutes		
VI_ATTR_MAINFRAME_LA	RO	Global	ViInt16	0 to 255 VI_UNKNOWN_LA	N/A
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_VXI_TRIG_STATUS	RO	Global	ViUInt32	N/A	N/A
VI_ATTR_VXI_TRIG_SUPPORT	RO	Global	ViUInt32	N/A	N/A
VI_ATTR_VXI_VME_INTR_STATUS	RO	Global	ViUInt16	N/A	N/A
VI_ATTR_VXI_VME_SYSFAIL_STATE	RO	Global	ViInt16	VI_STATE_ASSERTED VI_STATE_UNASSERTED VI_STATE_UNKNOWN	N/A

BACKPLANE Resource Attribute Descriptions

Attribute Name	Description

Generic BACKPLANE Resource Attributes

VI_ATTR_INTF_INST_NAME	Human-readable text describing the given interface.
VI_ATTR_INTF_NUM	Board number for the given interface.
VI_ATTR_INTF_TYPE	Interface type of the given session.
VI_ATTR_TMO_VALUE	Minimum timeout value to use, in milliseconds. A timeout value of VI_TMO_IN means that operations should never wait for the device to respond. A timeout value VI_TMO_INFINITE disables the timeout mechanism.
VXI and GPIB-VXI Specific BACKPL	ANE Resource Attributes
νι αττρ μαινεράμε ι α	This is the logical address of a given device in the mainframe, usually the device v 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 VIS. can choose any of these values, but must be consistent across mainframes. If this v known, the attribute value returned is VI_UNKNOWN_LA.
VI_ATTR_TRIG_ID	Identifier for the current triggering mechanism.
VI_ATTR_VXI_TRIG_STATUS	This attribute shows the current state of the VXI trigger lines. This is a bit vector v 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 with bits 0-9 corresponding to VI_TRIG_TTL0 through VI_TRIG_ECL1. Agilent returns 12 to indicate VI_TRIG_PANEL_IN for received triggers and VI_TRIG_F for asserted triggers on Agilent VXI controllers.
VI_ATTR_VXI_VME_INTR_STATUS	This attribute shows the current state of the VXI/VME interrupt lines. This is a bit bits 0-6 corresponding to interrupt lines 1-7.
VI_ATTR_VXI_VME_SYSFAIL_STATE	This attribute shows the current state of the VXI/VME SYSFAIL (SYStem FAILu line.

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 ca are VXI hardware triggers on the assertion edge (SYNC and ON trigger protocols only).

Event Attribute	Description	AP	Туре	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_ VI_TRIG_ECL0 to VI_TRIG_
VI_EVENT_VXI_VME_SYSFA	IL - Notification that the VXI/VME SY	SFAII	.* line has bee	n asserted.
Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_VXI_VME_SYS

VI_EVENT_VXI_VME_SYSRESET - Notification that the VXI/VME SYSRESET* line has been reset.

Event Attribute	Description	AP 7	Гуре	Range

VI_ATTR_EVENT_TYPE Unique logical identifier of the event. RO ViEventType VI_EVENT_VXI_VME_SYS

BACKPLANE Resource Operations

viAssertTrigger (vi, protocol)

viAssertUtilSignal(vi, line)

viAssertIntrSignal(vi, mode, statusID)
viMapTrigger (vi, trigSrc, trigDest, mode)
viUnmapTrigger (vi, trigSrc, trigDest)

Memory Access (MEMACC) Resource

This topic 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 <u>viSetAttribute</u>.

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

Note: AP = Access Privileges

Attribute Name	AP		Data Type	Range	Defau
Generic MFMACC Resource Attribu	ites				
VI_ATTR_DMA_ALLOW_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE	N/A
VI_ATTR_INTF_INST_NAME	RO	Global	ViString	N/A	N/A
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 VI_INTF_PXI	N/A
VI_ATTR_TMO_VALUE	RW	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to FFFFFFE _h VI_TMO_INFINITE	2000 mse
VXI and GPIB-VXI and PXI Specific	MEM	ACC Resou	rce Attributes		
VI_ATTR_DEST_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_SRC_INCREMENT	RW	Local	ViInt32	0 to 1	1
VI_ATTR_WIN_ACCESS	RO	Local	ViUInt16	VI_NMAPPED VI_USE_OPERS VI_DEREF_ADDR	VI_NMA
VI_ATTR_WIN_BASE_ADDR_32	RO	Local	ViBusAddress	N/A	N/A
VI_ATTR_WIN_BASE_ADDR_64	RO	Local	ViBusAddress64	N/A	N/A
VI_ATTR_WIN_SIZE_32	RO	Local	ViBusSize	N/A	N/A
VI_ATTR_WIN_SIZE_64	RO	Local	ViBusSize64	N/A	N/A
VXI and GPIB-VXI Specific MEMA	CC Res	ource Attrib	outes		
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_PRIV VI_D64_PRIV	VI_DATA

VI_ATTR_DEST_BYTE_ORDER RW Local ViUInt16 VI_BIG_ENDIAN VI_BIG_ VI_LITTLE_ENDIAN VI_BIG_ VI_DATA_NPRIV VI_DATA_PRIV

VI_ATTR_SRC_ACCESS_PRIV	RW	Local	ViUInt16	VI_PROG_NPRIV VI_PROG_PRIV VI_BLCK_NPRIV VI_BLCK_PRIV VI_D64_NPRIV VI_D64_PRIV	VI_DATA
VI_ATTR_SRC_BYTE_ORDER	RW	Local	ViUInt16	VI_BIG_ENDIAN VI_LITTLE_ENDIAN	VI_BIG_
VI_ATTR_VXI_LA	RO	Global	ViInt16	0 to 255	N/A
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
VI_ATTR_WIN_BYTE_ORDER	RW*	Local	ViUInt16	VI_BIG_ENDIAN VI LITTLE ENDIAN	VI_BIG_

* For VISA 2.2, the attributes VI_ATTR_WIN_BYTE_ORDER and VI_ATTR_WIN_ACCESS_PRIV are RW (readable and when the corresponding session is not mapped (VI_ATTR_WIN_ACCESS = = VI_NMAPPED). When the session is mappe attributes are RO (read only).

VXI and GPIB-VXI Specific MEMACC Resource Attributes

VI_ATTR_GPIB_SECONDARY_ADD	R RO	Global	ViUInt16	0 to 30, VI_NO_SEC_ADDR	N/A
VI_ATTR_GPIB_PRIMARY_ADDR	RO	Global	ViUInt16	0 to 30	N/A
VI_ATTR_INTF_PARENT_NUM	RO	Global	ViUInt16	0 to FFFF _h	N/A

MEMACC Resource Attribute Descriptions

E VI_ locum lor is a
MO_I eout va
stination hat is, tXX o ation y
ce offs hat is, htion n ion wi
oed, th derefe

VXI and GPIB-VXI Specific MEMACC Resource Attributes

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_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_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_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_VXI_LA	Logical address of the local controller.

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.			
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.			
GPIB-VXI Specific MEMACC Resource Attributes				
VI_ATTR_GPIB_PRIMARY_ADDF	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.

VI_ATTR_INTF_PARENT_NUM Board number of the GPIB board to which the GPIB-VXI is attached.

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	<u>AP</u>	Data Type	Range
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_COMPLETION
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	iJobId	N/A
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	iString	N/A
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed.	RO	iStatus	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViBus Size	*
VI_ATTR_RET_COUNT_32	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to FFFFFFFFh
VI_ATTR_RET_COUNT_64**	Actual number of elements that were asynchronously transferred.	RO	ViUInt64	$0\ \text{to}\ \text{FFFFFFF}\ \text{FFFFFF}\ h$

*The data type is defined in the appropriate VPP 4.3.x framework specification. **Defined only for operating systems that are 64-bit native.

MEMACC Resource Operations

viIn8 (vi, space, offset, val8) viIn16 (vi, space, offset, val16) viIn32 (vi, space, offset, val32) viIn64 (vi, space, offset, val64)

viMapAddress (vi, mapSpace, mapBase, mapSize, access, suggested, address) **viMapAddressEx** (vi, mapSpace, mapBase64, mapSize, access, suggested, address)

viMemAlloc(vi, size, offset) viMemFree(vi, offset) viMemAllocEx(vi, size, offset64) viMemFreeEx(vi, offset64) viMove (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length) viMovEx (vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length) viMoveAsync (vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId) viMoveAsyncEx (vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId)

viMoveIn8 (vi, space, offset, length, buf8) viMoveIn16 (vi, space, offset, length, buf16) viMoveIn32 (vi, space, offset, length, buf32) viMoveIn64 (vi, space, offset64, length, buf64) viMoveIn8Ex (vi, space, offset64, length, buf16) viMoveIn16Ex (vi, space, offset64, length, buf16) viMoveIn32Ex (vi, space, offset64, length, buf32) viMoveIn64Ex (vi, space, offset64, length, buf64) viMoveOut8 (vi, space, offset, length, buf8) viMoveOut8 (vi, space, offset, length, buf16) viMoveOut16 (vi, space, offset, length, buf16) viMoveOut32 (vi, space, offset, length, buf16) viMoveOut64 (vi, space, offset, length, buf32) viMoveOut64 (vi, space, offset, length, buf32) viMoveOut16Ex (vi, space, offset64, length, buf16) viMoveOut32Ex (vi, space, offset64, length, buf32) viMoveOut64Ex (vi, space, offse64, length, buf64)

viOut8 (vi, space, offset, val8) viOut16 (vi, space, offset, val16) viOut32 (vi, space, offset, val32) viOut64 (vi, space, offset, val64)

viPeek8 (vi, addr, val8) viPeek16 (vi, addr, val16) viPeek32 (vi, addr, val32) viPeek64 (vi, addr, val64) viPoke8 (vi, addr, val8) viPoke16 (vi, addr, val16) viPoke32 (vi, addr, val32) viPoke64 (vi, addr, val64)

viUnmapAddress (vi)

VISA Attribute Codes

The following table lists all of the VISA Attributes along with their Hex and Decimal codes.

Attribute Name	Hex Value	Decimal Value	Attribute Name
VI_ATTR_4882_COMPLIANT	3FFF019Fh	1073676703	VI_ATTR_PXI_MEM_SIZE_BAR1
VI_ATTR_ASRL_AVAIL_NUM	3FFF00ACh	1073676460	VI_ATTR_PXI_MEM_SIZE_BAR2
VI_ATTR_ASRL_BAUD	3FFF0021h	1073676321	VI_ATTR_PXI_MEM_SIZE_BAR3
VI_ATTR_ASRL_CTS_STATE	3FFF00AEh	1073676462	VI_ATTR_PXI_MEM_SIZE_BAR4
VI_ATTR_ASRL_DATA_BITS	3FFF0022h	1073676322	VI_ATTR_PXI_MEM_SIZE_BAR5
VI_ATTR_ASRL_DCD_STATE	3FFF00AFh	1073676463	VI_ATTR_PXI_MEM_TYPE_BAR0
VI_ATTR_ASRL_DSR_STATE	3FFF00B1h	1073676465	VI_ATTR_PXI_MEM_TYPE_BAR1
VI_ATTR_ASRL_DTR_STATE	3FFF00B2h	1073676466	VI_ATTR_PXI_MEM_TYPE_BAR2
VI_ATTR_ASRL_END_IN	3FFF00B3h	1073676467	VI_ATTR_PXI_MEM_TYPE_BAR3
VI_ATTR_ASRL_END_OUT	3FFF00B4h	1073676468	VI_ATTR_PXI_MEM_TYPE_BAR4
VI_ATTR_ASRL_FLOW_CNTRL	3FFF0025h	1073676325	VI_ATTR_PXI_MEM_TYPE_BAR5
VI_ATTR_ASRL_PARITY	3FFF0023h	1073676323	VI_ATTR_PXI_SLOT_LBUS_LEFT
VI_ATTR_ASRL_REPLACE_CHAR	3FFF00BEh	1073676478	VI_ATTR_PXI_SLOT_LBUS_RIGHT
VI_ATTR_ASRL_RI_STATE	3FFF00BFh	1073676479	VI_ATTR_PXI_SLOT_LWIDTH
VI_ATTR_ASRL_RTS_STATE	3FFF00C0h	1073676480	VI_ATTR_PXI_SLOTPATH
VI_ATTR_ASRL_STOP_BITS	3FFF0024h	1073676324	VI_ATTR_PXI_STAR_TRIG_BUS
VI_ATTR_ASRL_XON_CHAR	3FFF00C1h	1073676481	VI_ATTR_PXI_STAR_TRIG_LINE
VI_ATTR_ASRL_XOFF_CHAR	3FFF00C2h	1073676482	VI_ATTR_PXI_TRIG_BUS
VI_ATTR_BUFFER	3FFF4027h	1073692711	VI_ATTR_RD_BUF_OPER_MODE
VI_ATTR_CMDR_LA	3FFF006Bh	1073676395	VI_ATTR_RD_BUF_SIZE
VI_ATTR_DEST_ACCESS_PRIV	3FFF0039h	1073676345	VI_ATTR_RECV_INTR_LEVEL
VI_ATTR_DEST_BYTE_ORDER	3FFF003Ah	1073676346	VI_ATTR_RECV_TCPIP_ADDR
VI_ATTR_DEST_INCREMENT	3FFF0041h	1073676353	VI_ATTR_RECV_TRIG_ID
VI_ATTR_DEV_STATUS_BYTE	3FFF0189h	1073676681	VI_ATTR_RET_COUNT

VI_ATTR_DMA_ALLOW_EN	3FFF001Eh	1073676318	VI_ATTR_RET_COUNT_32
VI_ATTR_EVENT_TYPE	3FFF4010h	1073692688	VI_ATTR_RET_COUNT_64
VI_ATTR_FDC_CHNL	3FFF000Dh	1073676301	VI_ATTR_RM_SESSION
VI_ATTR_FDC_GEN_SIGNAL_EN	3FFF0011h	1073676305	VI_ATTR_RSRC_CLASS
VI_ATTR_FDC_MODE	3FFF000Fh	1073676303	VI_ATTR_RSRC_IMPL_VERSION
VI_ATTR_FDC_USE_PAIR	3FFF0013h	1073676307	VI_ATTR_RSRC_LOCK_STATE
VI_ATTR_FILE_APPEND_EN	3FFF0192h	1073676690	VI_ATTR_RSRC_MANF_ID
VI_ATTR_GPIB_ADDR_STATE	3FFF005Ch	1073676380	VI_ATTR_RSRC_MANF_NAME
VI_ATTR_GPIB_ATN_STATE	3FFF0057h	1073676375	VI_ATTR_RSRC_NAME
VI_ATTR_GPIB_CIC_STATE	3FFF005Eh	1073676382	VI_ATTR_RSRC_SPEC_VERSION
VI_ATTR_GPIB_HS488_CBL_LEN	3FFF0069h	1073676393	VI_ATTR_SEC_ACCESS_PRIV
VI_ATTR_GPIB_NDAC_STATE	3FFF0062h	1073676386	VI_ATTR_SEC_INCREMENT
VI_ATTR_GPIB_PRIMARY_ADDR	3FFF0172h	1073725810	VI_ATTR_SEND_END_EN
VI_ATTR_GPIB_READDR_EN	3FFF001Bh	1073676315	VI_ATTR_SIGP_STATUS_ID
VI_ATTR_GPIB_REN_STATE	3FFF0181h	1073676673	VI_ATTR_SLOT
VI_ATTR_GPIB_SECONDARY_ADDR	3FFF0173h	1073676659	VI_ATTR_SRC_BYTE_ORDER
VI_ATTR_GPIB_SRQ_STATE	3FFF0067h	1073676391	VI_ATTR_STATUS
VI_ATTR_GPIB_SYS_CNTRL_STATE	3FFF0068h	1073676392	VI_ATTR_SUPPRESS_END_EN
VI_ATTR_GPIB_UNADDR_EN	3FFF0184h	1073676676	VI_ATTR_TCPIP_ADDR
VI_ATTR_GRIB_RECV_CIC_STATE	3FFF4193h	1073693075	VI_ATTR_TCPIP_DEVICE_NAME
VI_ATTR_IMMEDIATE_SERV	3FFF0100h	1073676544	VI_ATTR_TCPIP_HISLIP_OVERLAP_EN
VI_ATTR_INTF_INST_NAME	BFFF00E9h	-1073807127	VI_ATTR_TCPIP_HISLIP_VERSION
VI_ATTR_INTF_NUM	3FFF0176h	1073676662	VI_ATTR_TCPIP_HISLIP_MAX_MESSAGE_F
VI_ATTR_INTF_PARENT_NUM	3FFF0101h	1073676545	VI_ATTR_TCPIP_HOSTNAME
VI_ATTR_INTF_TYPE	3FFF0171h	1073676657	VI_ATTR_TCPIP_IS_HISLIP
VI_ATTR_INTR_STATUS_ID	3FFF4023h	1073692707	VI_ATTR_TCPIP_KEEPALIVE
VI_ATTR_IO_PROT	3FFF001Ch	1073676316	VI_ATTR_TCPIP_NODELAY
VI_ATTR_JOB_ID	3FFF4006h	1073692678	VI_ATTR_TCPIP_PORT
VI_ATTR_MAINFRAME_LA	3FFF0070h	1073676400	VI_ATTR_TERMCHAR
VI_ATTR_MANF_ID	3FFF00D9h	1073676505	VI_ATTR_TERMCHAR_EN
VI_ATTR_MANF_NAME	BFFF0072h	-1073807246	 VI_ATTR_TMO_VALUE
VI_ATTR_MAX_QUEUE_LENGTH	3FFF0005h	1073676293	 VI_ATTR_TRIG_ID

VI_ATTR_MEM_BASE_32	3FFF00ADh	1073676461	VI_ATTR_USB_INTFC_NUM
VI_ATTR_MEM_BASE_64	3FFF00D0h	1073676496	VI_ATTR_USB_MAX_INTR_SIZE
VI_ATTR_MEM_SIZE_32	3FFF00DDh	1073676509	VI_ATTR_USB_PROTOCOL
VI_ATTR_MEM_SIZE_64	3FFF00D1h	1073676497	VI_ATTR_USB_RECV_INTR_DATA
VI_ATTR_MEM_SPACE	3FFF00DEh	1073676510	VI_ATTR_USB_RECV_INTR_SIZE
VI_ATTR_MODEL_CODE	3FFF00DFh	1073676511	VI_ATTR_USB_SERIAL_NUM
VI_ATTR_MODEL_NAME	BFFF0077h	-1073807241	VI_ATTR_USER_DATA
VI_ATTR_OPER_NAME	BFFF4042h	-1073790910	VI_ATTR_USER_DATA_32
VI_ATTR_PXI_ACTUAL_LWIDTH	3FFF0243h	1073676867	VI_ATTR_USER_DATA_64***
VI_ATTR_PXI_BUS_NUM	3FFF0205h	1073676805	VI_ATTR_VXI_DEV_CLASS
VI_ATTR_PXI_CHASSIS	3FFF0206h	1073676806	VI_ATTR_VXI_LA
VI_ATTR_PXI_DEV_NUM	3FFF0201h	1073676801	VI_ATTR_VXI_TRIG_STATUS
VI_ATTR_PXI_DSTAR_BUS	3FFF0244h	1073676868	VI_ATTR_VXI_TRIG_SUPPORT
VI_ATTR_PXI_DSTAR_SET	3FFF0245h	1073676869	VI_ATTR_VXI_VME_INTR_STATUS
VI_ATTR_PXI_FUNC_NUM	3FFF0202h	1073676802	VI_ATTR_VXI_VME_SYSFAIL_STATE
VI_ATTR_PXI_IS_EXPRESS	3FFF0240h	1073676864	VI_ATTR_WIN_ACCESS
VI_ATTR_PXI_MAX_LWIDTH	3FFF0242h	1073676866	VI_ATTR_WIN_ACCESS_PRIV
VI_ATTR_PXI_MEM_BASE_BAR0	3FFF0221h	1073676833	VI_ATTR_WIN_BASE_ADDR_32
VI_ATTR_PXI_MEM_BASE_BAR1	3FFF0222h	1073676834	VI_ATTR_WIN_BASE_ADDR_64
VI_ATTR_PXI_MEM_BASE_BAR2	3FFF0223h	1073676835	VI_ATTR_WIN_BYTE_ORDER
VI_ATTR_PXI_MEM_BASE_BAR3	3FFF0224h	1073676836	VI_ATTR_WIN_SIZE_32
VI_ATTR_PXI_MEM_BASE_BAR4	3FFF0225h	1073676837	VI_ATTR_WIN_SIZE_64
VI_ATTR_PXI_MEM_BASE_BAR5	3FFF0226h	1073676838	VI_ATTR_WR_BUF_OPER_MODE
VI_ATTR_PXI_MEM_SIZE_BAR0	3FFF0231h	1073676849	VI_ATTR_WR_BUF_SIZE

* For 32-bit systems, these values are equivalent to those for VI_ATTR_RET_COUNT_32. For 64-bit systems, these values are equivalent to those for VI_ATTR_RET_COUNT_64.

** For 32-bit systems, these values are equivalent to those for VI_ATTR_USER_DATA_32. For 64-bit systems, these values are equivalent to those for VI_ATTR_USER_DATA_64.

*** Defined only for frameworks that are 64-bit native.

VISA Type Definitions

This topic lists and describes the VISA data types.

VISA Data 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 <i>ViAddr</i> .
ViChar	char	An 8-bit integer representing an ASCII character.
ViPChar	ViChar *	The location of a <i>ViChar</i> .
ViByte	unsigned char	An 8-bit unsigned integer representing an extended ASCII character.
ViPByte	ViByte *	The location of a <i>ViByte</i> .
ViBoolean	ViUInt16	A type that is either <i>VI_TRUE</i> or <i>VI_FALSE</i> .
ViPBoolean	ViBoolean *	The location of a <i>ViBoolean</i> .
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 <i>ViVersion</i> .
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 <i>ViEventType</i> .
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 <i>ViObject</i> .
ViSession	ViObject	Specifies the information necessary to manage a communication channel with a resource.
ViPSession	ViSession *	The location of a ViSession.
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 <i>ViEvent</i> .
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

ViKeyId

ViString

job identifier.

The location of a string.

VISA Error Codes

This topic lists VISA error codes in numerical order and alphabetically by description.

Error Codes Listed in Numerical Order

Hex Value Decimal Value VISA Status Code

Success Codes

3FFF0002	1073676290	VI_SUCCESS_EVENT_EN
3FFF0003	1073676291	VI_SUCCESS_EVENT_DIS
3FFF0004	1073676292	VI_SUCCESS_QUEUE_EMPTY
3FFF0005	1073676293	VI_SUCCESS_TERM_CHAR
3FFF0006	1073676294	VI_SUCCESS_MAX_CNT
3FFF007D	1073676413	VI_SUCCESS_DEV_NPRESENT
3FFF007E	1073676414	VI_SUCCESS_TRIG_MAPPED
3FFF0080	1073676416	VI_SUCCESS_QUEUE_NEMPTY
3FFF0098	1073676440	VI_SUCCESS_NCHAIN
3FFF0099	1073676441	VI_SUCCESS_NESTED_SHARED
3FFF009A	1073676442	VI_SUCCESS_NESTED_EXCLUSIVE
3FFF009B	1073676443	VI_SUCCESS_SYNC

Warning Codes

3FFF000C	1073676300	VI_WARN_QUEUE_OVERFLOW
3FFF0077	1073676407	VI_WARN_CONFIG_NLOADED
3FFF0082	1073676418	VI_WARN_NULL_OBJECT
3FFF0084	1073676420	VI_WARN_NSUP_ATTR_STATE
3FFF0085	1073676421	VI_WARN_UNKNOWN_STATUS
3FFF0088	1073676424	VI_WARN_NSUP_BUF
3FFF00A9	1073676457	VI_WARN_EXT_FUNC_NIMPL

Error Codes

BFFF0000	-1073807360	VI_ERROR_SYSTEM_ERROR
BFFF000E	-1073807346	VI_ERROR_INV_OBJECT
BFFF000F	-1073807345	VI_ERROR_RSRC_LOCKED
BFFF0010	-1073807344	VI_ERROR_INV_EXPR
BFFF0011	-1073807343	VI_ERROR_RSRC_NFOUND
BFFF0012	-1073807342	VI_ERROR_INV_RSRC_NAME
BFFF0013	-1073807341	VI_ERROR_INV_ACC_MODE
BFFF0015	-1073807339	VI_ERROR_TMO
BFFF0016	-1073807338	VI_ERROR_CLOSING_FAILED
BFFF001B	-1073807333	VI_ERROR_INV_DEGREE
BFFF001C	-1073807332	VI_ERROR_INV_JOB_ID
BFFF001D	-1073807331	VI_ERROR_NSUP_ATTR
BFFF001E	-1073807330	VI_ERROR_NSUP_ATTR_STATE
BFFF001F	-1073807329	VI_ERROR_ATTR_READONLY
BFFF0020	-1073807328	VI_ERROR_INV_LOCK_TYPE
BFFF0021	-1073807327	VI_ERROR_INV_ACCESS_KEY
BFFF0026	-1073807322	VI_ERROR_INV_EVENT
BFFF0027	-1073807321	VI_ERROR_INV_MECH
BFFF0028	-1073807320	VI_ERROR_HNDLR_NINSTALLED
BFFF0029	-1073807319	VI_ERROR_INV_HNDLR_REF
BFFF002A	-1073807318	VI_ERROR_INV_CONTEXT
BFFF002F	-1073807313	VI_ERROR_NENABLED
BFFF0030	-1073807312	VI_ERROR_ABORT
BFFF0034	-1073807308	VI_ERROR_RAW_WR_PROT_VIOL
BFFF0035	-1073807307	VI_ERROR_RAW_RD_PROT_VIOL
BFFF0036	-1073807306	VI_ERROR_OUTP_PROT_VIOL
BFFF0037	-1073807305	VI_ERROR_INP_PROT_VIOL
BFFF0038	-1073807304	VI_ERROR_BERR
BFFF0039	-1073807303	VI_ERROR_IN_PROGRESS
BFFF003A	-1073807302	VI_ERROR_INV_SETUP
BFFF003B	-1073807301	VI_ERROR_QUEUE_ERROR
BFFF003C	-1073807300	VI_ERROR_ALLOC
BFFF003D	-1073807299	VI_ERROR_INV_MASK

BFFF003E	-1073807298	VI_ERROR_IO
BFFF003F	-1073807297	VI_ERROR_INV_FMT
BFFF0041	-1073807295	VI_ERROR_NSUP_FMT
BFFF0042	-1073807294	VI_ERROR_LINE_IN_USE
BFFF0046	-1073807290	VI_ERROR_NSUP_MODE
BFFF004A	-1073807286	VI_ERROR_SRQ_NOCCURRED
BFFF004E	-1073807282	VI_ERROR_INV_SPACE
BFFF0051	-1073807279	VI_ERROR_INV_OFFSET
BFFF0052	-1073807278	VI_ERROR_INV_WIDTH
BFFF0054	-1073807276	VI_ERROR_NSUP_OFFSET
BFFF0055	-1073807275	VI_ERROR_NSUP_VAR_WIDTH
BFFF0057	-1073807273	VI_ERROR_WINDOW_NMAPPED
BFFF0059	-1073807271	VI_ERROR_RESP_PENDING
BFFF005F	-1073807265	VI_ERROR_NLISTENERS
BFFF0060	-1073807264	VI_ERROR_NCIC
BFFF0061	-1073807263	VI_ERROR_NSYS_CNTLR
BFFF0067	-1073807257	VI_ERROR_NSUP_OPER
BFFF0068	-1073807256	VI_ERROR_INTR_PENDING
BFFF006A	-1073807254	VI_ERROR_ASRL_PARITY
BFFF006B	-1073807253	VI_ERROR_ASRL_FRAMING
BFFF006C	-1073807252	VI_ERROR_ASRL_OVERRUN
BFFF006E	-1073807250	VI_ERROR_TRIG_NMAPPED
BFFF0070	-1073807248	VI_ERROR_NSUP_ALIGN_OFFSET
BFFF0071	-1073807247	VI_ERROR_USER_BUF
BFFF0072	-1073807246	VI_ERROR_RSRC_BUSY
BFFF0076	-1073807242	VI_ERROR_NSUP_WIDTH
BFFF0078	-1073807240	VI_ERROR_INV_PARAMETER
BFFF0079	-1073807239	VI_ERROR_INV_PROT
BFFF007B	-1073807237	VI_ERROR_INV_SIZE
BFFF0080	-1073807232	VI_ERROR_WINDOW_MAPPED
BFFF0081	-1073807231	VI_ERROR_NIMPL_OPER
BFFF0083	-1073807229	VI_ERROR_INV_LENGTH
BFFF0091	-1073807215	VI_ERROR_INV_MODE
BFFF009C	-1073807204	VI_ERROR_SESN_NLOCKED
BFFF009D	-1073807203	VI_ERROR_MEM_NSHARED
BFFF009E	-1073807202	VI_ERROR_LIBRARY_NFOUND
BFFF009F	-1073807201	VI_ERROR_NSUP_INTR
----------	-------------	---------------------------
BFFF00A0	-1073807200	VI_ERROR_INV_LINE
BFFF00A1	-1073807199	VI_ERROR_FILE_ACCESS
BFFF00A2	-1073807198	VI_ERROR_FILE_IO
BFFF00A3	-1073807197	VI_ERROR_NSUP_LINE
BFFF00A4	-1073807196	VI_ERROR_NSUP_MECH
BFFF00A5	-1073807195	VI_ERROR_INTF_NUM_NCONFIG
BFFF00A6	-1073807194	VI_ERROR_CONN_LOST
BFFF00A7	-1073807193	VI_ERROR_MACHINE_NAVAIL
BFFF00A8	-1073807192	VI_ERROR_NPERMISSION

Alphabetical Description of Error Codes

VISA Status Codes

Description

Success Codes

VI_SUCCESS

VI_SUCCESS_DEV_NPRESENT

VI_SUCCESS_EVENT_DIS

VI_SUCCESS_EVENT_EN

VI_SUCCESS_MAX_CNT

VI_SUCCESS_NCHAIN

VI_SUCCESS_NESTED_EXCLUSIVE

VI_SUCCESS_NESTED_SHARED

VI_SUCCESS_QUEUE_EMPTY

VI_SUCCESS_QUEUE_NEMPTY

VI_SUCCESS_SYNC

VI_SUCCESS_TERM_CHAR

VI_SUCCESS_TRIG_MAPPED

Operation completed successfully.

Session opened successfully, but the device at the specified address is r responding.

The specified event is already disabled.

The specified event is already enabled for at least one of the specified mechanisms.

The number of bytes specified were read.

Event handled successfully. Do not invoke any other handlers on this so this event.

The specified access mode was successfully acquired and this session h exclusive locks.

The specified access mode was successfully acquired and this session **b** shared locks.

The event queue was empty while trying to discard queued events.

The event queue is not empty.

The read or write operation performed synchronously.

The specified termination character was read.

The path from *trigSrc* to *trigDest* is already mapped.

Warning Codes

The specified configuration either does not exist or could not be loaded VI_WARN_CONFIG_NLOADED 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_NULL_OBJECT The specified object reference is uninitialized. The device sent more data than the user specified in VI_WARN_QUEUE_OVERFLOW VI_ATTR_USB_MAX_INTR_SIZE VI_WARN_UNKNOWN_STATUS The status code passed to the function was unable to be interpreted. The operation succeeded, but a lower level driver did not implement th VI_WARN_EXT_FUNC_NIMPL functionality

Error Codes

VI_ERROR_ABORT

VI_ERROR_ALLOC

VI_ERROR_ASRL_FRAMING

VI_ERROR_ASRL_OVERRUN

VI ERROR ASRL PARITY

VI_ERROR_ATTR_READONLY

VI_ERROR_BERR

VI_ERROR_CLOSING_FAILED

VI_ERROR_CONN_LOST

VI_ERROR_FILE_ACCESS

VI_ERROR_FILE_IO

VI_ERROR_HNDLR_NINSTALLED

VI_ERROR_INP_PROT_VIOL

VI_ERROR_INTF_NUM_NCONFIG VI_ERROR_INTR_PENDING VI_ERROR_INV_ACC_MODE

VI ERROR INV ACCESS KEY

VI_ERROR_INV_CONTEXT

VI_ERROR_INV_DEGREE

VI_ERROR_INV_EVENT

VI_ERROR_INV_EXPR

VI_ERROR_INV_FMT

VI_ERROR_INV_HNDLR_REF

VI ERROR INV JOB ID

VI_ERROR_INV_LENGTH

VI_ERROR_INV_LINE

VI_ERROR_INV_LOCK_TYPE

VI_ERROR_INV_MASK

VI_ERROR_INV_MECH

Calls in the current process executing on the specified *vi* are aborted.

Insufficient system resources to open a session or to allocate the buffer memory block of the specified size.

A framing error occurred during transfer.

An overrun error occurred during transfer. A character was not read fro hardware before the next character arrived.

A parity error occurred during transfer.

The specified attribute is read-only.

A bus error occurred during transfer.

Unable to deallocate the previously allocated data structures for this session.

A TCP connection is dropped as a result of keep-alives packets.

An error occurred while trying to open the specified file. Possible reasc an invalid path or lack of access rights.

An error occurred while accessing the specified file.

A handler is not currently installed for the specified event. The session enabled for the VI_HNDLR mode of the callback mechanism.

Input protocol error occurred during transfer.

The interface type is valid but the specified interface number is not con

An interrupt is still pending from a previous call.

Invalid access mode.

The *requestedKey* value passed in is not a valid access key to the specific resource.

The event context specified is invalid.

The specified degree is invalid.

The event type specified is invalid for the specified resource.

The expression specified is invalid.

The format specifier is invalid for the current argument.

The specified handler reference and/or the user context value does not a installed handler.

The specified job identifier is invalid.

The length specified is invalid.

The value specified by the *line* parameter is invalid.

The specified type of lock is not supported by this resource.

The system cannot set the buffer for the given mask or the specified manot specify a valid flush operation on the read/write resource.

The mechanism specified for the event is invalid.

VI_ERROR_INV_MODE

VI_ERROR_INV_OBJECT VI_ERROR_INV_OFFSET VI_ERROR_INV_PARAMETER VI_ERROR_INV_PROT VI_ERROR_INV_RSRC_NAME VI_ERROR_INV_SESSION

VI_ERROR_INV_SETUP

VI_ERROR_INV_SIZE

VI_ERROR_INV_SPACE

VI_ERROR_INV_WIDTH

VI_ERROR_IO

VI_ERROR_LIBRARY_NFOUND

VI_ERROR_LINE_IN_USE

VI_ERROR_MACHINE_NAVAIL

VI_ERROR_MEM_NSHARED

VI_ERROR_NCIC

VI_ERROR_NENABLED

VI_ERROR_NIMPL_OPER

VI_ERROR_NLISTENERS

VI_ERROR_NPERMISSION

VI_ERROR_NSUP_ALIGN_OFFSET

VI_ERROR_NSUP_ATTR

VI_ERROR_NSUP_ATTR_STATE

VI_ERROR_NSUP_FMT

VI_ERROR_NSUP_INTR

VI_ERROR_NSUP_LINE

VI_ERROR_NSUP_MECH VI_ERROR_NSUP_MODE VI_ERROR_NSUP_OFFSET VI_ERROR_NSUP_OPER VI_ERROR_NSUP_VAR_WIDTH VI_ERROR_NSUP_WIDTH The value specified by the *mode* parameter is invalid.

The object reference is invalid.

The offset specified is invalid.

The value of some parameter is invalid.

The protocol specified is invalid.

The resources specified are invalid.

The session specified is invalid.

The setup specified is invalid, possibly due to attributes being set to an inconsistent state, or some implementation-specific configuration file is does not exist.

The specified size is invalid.

The address space specified is invalid.

Invalid source or destination width specified.

Could not perform read/write function because of an I/O error, or an un error occurred during transfer.

A code library required by VISA could not be located or loaded.

The specified trigger line is in use.

The specified machine is not available.

The device does not export any memory.

The session is referring to something other than the controller in charge

The session must be enabled for events of the specified type to receive

The given operation is not implemented.

No listeners are detected. (Both NRFD and NDAC are deasserted.)

You do not have permission to perform this operation.

The specified offset is not properly aligned for the access width of the c

The attribute specified is not supported by the specified resource.

The state specified for the attribute is not supported.

The format specifier is not supported for the current argument type.

The interface cannot generate an interrupt on the requested level or wit requested *statusID* value.

One of the specified lines (*trigSrc* or *trigDest*) is not supported by this implementation.

The specified mechanism is not supported for the given event type.

The specified mode is not supported by this VISA implementation.

The offset specified is not accessible.

The operation specified is not supported in the given session.

Cannot support source and destination widths that are different.

The specified width is not supported by this hardware.

VI_ERROR_NSYS_CNTLR

VI_ERROR_OUTP_PROT_VIOL VI_ERROR_QUEUE_ERROR VI_ERROR_RAW_RD_PROT_VIOL VI_ERROR_RAW_WR_PROT_VIOL VI_ERROR_RESP_PENDING VI_ERROR_RSRC_BUSY VI_ERROR_RSRC_LOCKED VI_ERROR_RSRC_NFOUND

VI_ERROR_SESN_NLOCKED

VI_ERROR_SRQ_NOCCURED VI_ERROR_SYSTEM_ERROR

VI_ERROR_TMO

VI_ERROR_TRIG_NMAPPED

VI_ERROR_USER_BUF

VI_ERROR_WINDOW_MAPPED

VI_ERROR_WINDOW_NMAPPED

VI_ERROR_IN_PROGRESS

The interface associated with this session is not the system controller.

Output protocol error occurred during transfer.

Unable to queue read or write operation.

A violation of raw read protocol occurred during a transfer.

A violation of raw write protocol occurred during a transfer.

A previous response is still pending, causing a multiple query error.

The resource is valid, but VISA cannot currently access it.

The specified operation could not be performed because the resource ic *vi* has been locked for this kind of access.

The expression specified does not match any device, or resource was no

The current session did not have any lock on the resource.

A service request has not been received for the session.

Unknown system error.

The operation failed to complete within the specified timeout period.

The path from *trigSrc* to *trigDest* is not currently mapped.

A specified user buffer is not valid or cannot be accessed for the require

The specified session already contains a mapped window.

The specified session is not currently mapped.

Unable to start a new asynchronous operation while another asynchronoperation is in progress

VISA Directories

This topic provides information about the location of VISA software files.

Note: Always use "Add/Remove Programs" from the Windows Control Panel to remove the Agilent IO Libraries Suite; never delete installed files manually.

Windows Directory Structure

Agilent IO Libraries Suite 16 uses the new IVI VISA COM Standard Components. They install both the IVI VISA COM files, and create the VXI*plug&play* directory structure if it doesn't already exist. The default base directory for the VX*Iplug&play* components in this installer is:

C:\Program Files\IVI Foundation\VISA

Note that if you already have a set of VXIpnp directories on your PC, the new installer will continue to use your current set of directories -- probably either

C:\Program Files\VISA

or

C:\VXIPNP

and not create a new set in

C:*Program Files**IVI Foundation**VISA*.

The *visa32.dll*file (for 32- and 64-bit operating systems) and *visa64.dll* file (for 64-bit operating systems), are stored in the *C:\Windows\System32* folder. The VISA path can be displayed by clicking the IO icon (1) in the Windows notification area. Then click **Installation Information** to view a dialog box that contains the VISA path information. A typical display follows.

Note: Your Agilent IO Libraries Suite Version number may be different and your paths may be different if you did a custom installation.

Using the VISA C API in Microsoft Visual Basic 6

Using VISA to communicate with instruments using the VISA C API in Microsoft Visual Basic 6 (VB6) is relatively straightforward for most of the VISA functions. However, there are several areas of I/O which can cause difficulty. This topic provides details on those areas.

Include visa32.bas

To use the VISA C API in VB6, include the visa32.bas file in your projects. If you installed the Agilent IO Libraries Suite using the default VISA path. Refer to <u>VISA Directories</u> topic for specific information)..

We strongly recommend copying visa32.bas to your local project directory, so that any modifications you may make to it will be local to your project and won't change the master installed copy of that file. Once you have placed visa32.bas in your project directory, you will need to use the **Project** > **Add Module** menu choice to add visa32.bas to your project. Once this module is added, you will be able to call VISA functions from your VB6 project.

viRead and viWrite

The **viRead** and **viWrite** function declarations use the String data type for their buffer pointers, but users often want to read and write numeric array data. See <u>Notes on Using viRead/viWrite in Visual Basic 6</u> for more details and examples.

Formatted I/O

The **viPrintf**, **viScanf**, and **viQueryf** VISA functions take variable argument lists in C. Some of **viScanf**'s and **viQueryf**'s variable arguments are references to primitive (byte, integer, long, float, etc.) types, meaning that the values themselves can be changed by the function. There is no equivalent in VB6 for variable argument lists with reference arguments, so no direct translation is available. Instead we use the 'V' form of these functions (**viVPrintf**, **viVScanf**, and **viVQueryf**), in conjunction with the undocumented VB6 **VarPtr** function to create an array of pointers to arguments than can be passed to the VISA functions.

The examples below are presented as self-contained VB6 Sub's with comments explaining the various features. You can modify and adapt the code in the examples to your specific situation.

- <u>viVPrintf</u>/viVScanf Example using String
- **<u>viVScanf</u>** Example Returning a Double Array
- <u>viVScanf</u> Example Reading an IEEE 488 Definite Length Block and Returning a Byte Array
- <u>viVScanf</u> Example Reading an IEEE 488 Indefinite Length Block and Returning a Byte Array
- <u>viVQueryf</u> Example with String and Indefinite Length Block

See Also

viRead, viWrite, viVPrintf, viVScanf, viVQueryfVB6 Types and VISA C Types Reference Notes on Using viRead/viWrite in Visual Basic 6 viVPrintf/viVScanf Example using String viVScanf Example Returning a Double Array viVScanf Example Reading an IEEE 488 Definite Length Block and Returning a Byte Array viVScanf Example Reading an IEEE 488 Indefinite Length Block and Returning a Byte Array viVQueryf Example with String and Indefinite Length Block

VB6 Types and VISA C Types Reference

VISA Type	VB6 Type	Description
ViInt8, ViUInt8, ViChar, ViByte	Byte	
ViInt16, ViUInt16	Integer	Since VB6 does not support signed bytes, the byte type is used for both.
ViInt16, ViUInt16	Long	Since unsigned shorts are not supported in VB6, the signed type is used for both.
ViInt32, ViUInt32	Long	Since unsigned integers are not supported in VB6, the signed type is used for both.
ViSession, ViPFindList, ViPEvent	Long	Sessions and other VISA object handles are just 32-bit integers.
ViStatus	Long	Status codes (error codes) have always been 32-bit integers.
ViAddr	Long	This is a reference to a remote 32-bit memory space, and will therefore fit in a 32-bit integer
ViBuf	String	This type represents byte buffers or ASCII strings that are not to be modified by VISA. The String type was chosen for methods that typically write ANSI strings. When a string is passed 'byVal' to an external DLL (such as visa32.dll) VB6 automatically marshals the String type into a pointer to an ASCII string that VISA can accept.

ViPBuf	String	This type represents byte buffers or ASCII string buffers that are to be written to by VISA. The String type was chosen for functions that typically write ASCII strings, because the default marshaling behavior of that class is to give the C function a pointer to the beginning of the preallocated buffer, and to use the 0 ASCII value to determine the end of the string written in the buffer. Don't forget to allocate enough storage space in your String or to declare the String as fixed length (.e.g. <i>Dim myString as</i> <i>String</i> * 256) before calling the VISA function.
ViChar[]	String	Arrays of ViChar are used when VISA plans to write short- length strings into the passed-in buffer. Again, the String type is well-suited to acting as a character buffer. Don't forget to allocate enough storage space in your String or to declare the String as fixed length (e.g. <i>Dim myString as</i> <i>String</i> * 256) before calling the VISA function.
ViAttrState	Byte, Integer Long	The value passed to viSetAttribute is an 8-, 16-, or 32-bit integer.
void *	Byte, Integer Long String	viSetAttribute can return 8-, 16-, or 32-bit integers, or strings. Don't forget to allocate enough storage space in your String or to declare the string as fixed length (e.g. <i>Dim myString as String * 256</i>) before calling the VISA function.
ViPUInt8, ViPInt8, etc.	Byte, Integer, etc.	The ViPXXXX types where 'XXXXX' is an integral type are return values for integer types. Functions that return pointers to integer types have these parameters declared using the 'ByRef' keyword so the address of the variable is passed rather than its value.

See Also

Using the VISA C API in Microsoft Visual Basic 6

Using the VISA C API in Microsoft .NET

Test and measurement programmers accustomed to using the VISA C API (implemented by the visa32.dll C DLL) for communicating with instruments are familiar with the visa32.h C standard header file for use in C and C++. This header file is defined by the VXI*plug&play* Systems Alliance and distributed by Agilent Technologies, among others. The alliance also defines a header file VISA32.bas for Microsoft Visual Basic 6. However, there are at present no officially defined header files for programming in the VISA C API in the Microsoft .NET technology languages, such as C# and Visual Basic .NET.

This document describes Agilent's .NET header files for VISA, VISA32.cs for C# and VISA32.vb for VB .NET; provides examples for some of the methods; and provides a tutorial for using these files. Agilent defines and provides these files to allow programmatic access to the VISA C API from the two most popular .NET languages.

Programmers wishing to use the VISA C API in .NET should include the appropriate file in their project. The compiled .NET assembly will then have all the information it needs to use the VISA C Library (visa32.dll.) The <u>Tutorial</u> shows how to include and compile a program using these header files.

This help is broken into several topics following the introduction. The first, <u>VISA API Essentials</u>, provides examples and brief descriptions of the core set of methods you need to communicate with instruments through Agilent's .NET header files for VISA. The second section, <u>Tutorial</u>, provides a brief tutorial in both C# and VB .NET describing a complete use of these files from start to finish. The third section, <u>Advanced Use of viPrintf/viScanf</u>, describes how to add parameters and make your own versions of these functions. The last section, <u>.NET Types and VISA C Types Reference</u>, describes the translation of the C VISA data types to .NET and provides a conversion table.

VISA_API_Essentials

This section reviews the most essential VISA functions and provides examples of how to use them in .NET. Consult the <u>VISA Functions</u> volume of this help for a complete VISA reference.

- viOpenDefaultRM
- viOpen
- viClose
- <u>viRead</u>
- <u>viWrite</u>
- <u>viPrintf</u>
- •
- viScanf

00

Syntax

```
viOpen(int sesn, string viDesc, int mode, int timeout,
    out int vi);
```

Description

Opens a VISA resource session given a VISA Resource Manager session, a resource address, and resource locking information. If successful, it returns a session identifier (integer) that can be used to execute any other legal operations on that resource.

Parameters

Name	Dir	Туре	Description
sesn	IN	int	Resource Manager session (should always be the Default Resource Manager for VISA returned from <u>viOpenDefaultRM</u>).
viDesc	IN	string	Unique symbolic name of a resource. (See the viOpen topic in the VISA Help for details)
mode	IN	int	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	IN	int	If the <i>accessMode</i> parameter requires a lock, this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. Otherwise, this parameter is ignored. (Must use VI_NULL in VISA 1.0.)
vi	OUT	out int	Unique logical identifier reference to a session.

Return Values

Completion Codes	Description	
VI_SUCCESS	Operation completed 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_ALLOC	Insufficient system resources to open a session.	
VI_ERROR_INTF_NUM_NCONFIG	The interface type is valid but the specified interface number is not configured.	
VI_ERROR_INV_ACC_MODE	Invalid access mode.	
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.	
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).	
VI_ERROR_LIBRARY_NFOUND	A code library required by VISA could not be located or loaded.	
VI_ERROR_NSUP_OPER	The given <i>sesn</i> does not support this function. For VISA, this function is supported only by the Default Resource Manager 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_RSRC_NFOUND	Insufficient location information or resource not present in the system.	
VI_ERROR_TMO	A session to the resource could not be obtained within the specified <i>timeout</i> period.	

C# Example

VB.NET Example

Public Function OpenSession(ByVal resourceAddress As String, _ ByVal resourceManager As Integer) As Integer Dim session As Integer = 0, viError As Integer viError = visa32.viOpen(resourceManager, resourceAddress, _ visa32.VI_NO_LOCK, _ visa32.VI_TMO_IMMEDIATE, session) If viError < visa32.VI_SUCCESS Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(resourceManager, viError, err) Throw New ApplicationException(err.ToString()) End If Return session End Function

Syntax

Syntax

viClose(int vi);

Description

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. **Failure to close VISA objects will result in memory and resource leaks.**

Parameters

Name	Dir	Туре	Description
vi	IN	int	Unique logical identifier of a session, event, or find list.

Return Values

Completion Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_WARN_NULL_OBJECT	The specified object reference is uninitialized.
Error Codes	Description
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).

C# Example

```
public void CloseSession(int session)
{
     visa32.viClose(session);
}
```

VB .NET Example

Public Sub CloseSession(ByVal session As Integer) visa32.viClose(session) End Sub

viRead

Syntax

```
viRead(int vi, byte[] buffer, int count, out int
   retCount);
```

Description

Synchronously transfers data from a device. The data that is read is stored in the buffer represented by *buffer*. 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. It is possible to have one, two, or all three of these conditions satisfied at the same time.

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

Parameters

Name	Dir	Туре	Description
vi	IN	int	Unique logical identifier to a session.
buffer	OUT	byte[]	The array of bytes to receive data from device.
count	IN	int	Number of bytes to be read.
retCount	OUT	out int	Represents the location of an integer that will be set to the number of bytes actually transferred.

Return Values

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

C# Example

```
public int ReadBytes(int session, int maxCount, out byte[] data)
{
    data = new Byte[maxCount];
    int viError, readCount;
    viError = visa32.viRead(session, data, maxCount, out readCount);
    if (viError < visa32.VI_SUCCESS)
    {
        System.Text.StringBuilder error =
            new System.Text.StringBuilder(256);
        visa32.viStatusDesc(session, viError, error);
        throw new ApplicationException(error.ToString());
    }
    return readCount;
}
</pre>
```

VB.NET Example

Public Function ReadBytes(ByVal session As Integer, ByVal maxCount As Integer ByRef data() As Byte) As Integer data = New Byte(maxCount) {} Dim viError As Integer, readCount As Integer viError = visa32.viRead(session, data, maxCount, readCount) If viError < visa32.VI_SUCCESS Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(session, viError, err) Throw New ApplicationException(err.ToString()) End If Return readCount End Function
Syntax

Syntax

```
viWrite(int vi, byte[] buffer, int count, out int
  retCount);
```

Description

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

Name	Dir	Туре	Description
vi	IN	int	Unique logical identifier to a session.
buffer	IN	byte[]	Represents the location of a data block to be sent to device.
count	IN	int	Specifies number of bytes to be written.
retCount	OUT	out int	Represents the location of an integer that will be set to the number of bytes actually transferred.

Return Values

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

C# Example

```
public int WriteBytes(int session, int requestCount, byte[] data)
{
    int viError, outCount;
    viError = visa32.viWrite(session, data, requestCount, out outCount);
    if (viError < visa32.VI_SUCCESS)
    {
        System.Text.StringBuilder error =
            new System.Text.StringBuilder(256);
        visa32.viStatusDesc(session, viError, error);
        throw new ApplicationException(error.ToString());
    }
    return outCount;
}</pre>
```

VB .NET Example

Public Function WriteBytes(ByVal session As Integer, _ ByVal requestCount As Integer, _ ByVal data() As Byte) As Integer Dim viError As Integer, outCount As Integer = 0 viError = visa32.viWrite(session, data, requestCount, outCount) If viError < visa32.VI_SUCCESS Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(session, viError, err) Throw New ApplicationException(err.ToString()) End If Return outCount End Function

Syntax

Syntax

```
viPrintf(int vi, string writeFmt, <overloaded
    arguments>);
```

Description

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

For information about getting more options in the parameter list, see <u>Advanced</u> <u>Use of viPrintf/viScanf</u>.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	int	Unique logical identifier to a session.
writeFmt	IN	string	String describing the format for arguments.
overloaded argument	IN	N/A	A parameter the format string is applied to.

Return Values

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

C# Example

```
// Writes a comma-separated list of NR1-style integers to the formatted I/O
// buffer and flushes the buffer (because of the "\n".)
public void WriteCommaSeparatedInt32List(int session, int[] data, int count)
{
    string format = "%dl," + count + "\n";
    int viError;
    viError = visa32.viPrintf(session, format, data);
    if (viError < visa32.VI_SUCCESS)
    {
        System.Text.StringBuilder error =
            new System.Text.StringBuilder(256);
        visa32.viStatusDesc(session, viError, error);
        throw new ApplicationException(error.ToString());
    }
}</pre>
```

VB.NET Example

' Writes a comma-separated list of NR1-style integers to the formatted I/O ' buffer and flushes the buffer (because of the "\n".) Public Sub WriteCommaSeparatedInt32List(ByVal session As Integer, _ ByVal data() As Integer, _ ByVal count As Integer) Dim format As String = "%," & count & "d" & vbLf Dim viError As Integer viError = visa32.viPrintf(session, format, data) If viError < visa32.VI_SUCCESS Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(session, viError, err) Throw New ApplicationException(err.ToString()) End If End Sub

Tutorial

This tutorial will show you how to:

- Add Agilent's VISA header file for C# or VB .NET to your C# or VB .NET project
- Use the functions defined in the header file for communicating with an instrument
- Deal with .NET-specific issues that arise when using the VISA C API

The topics in the tutorial are:

- <u>Getting Started</u>
- <u>Creating Your Project</u>
- Adding Agilent's VISA Header Files to Your Project
- Using the Header File in Your Project
- Adding Error Handling to Your Project
- Reading and Writing Array Data
- Putting it All Together
- Deploying Your Project

Getting Started with VISA

The VISA C DLL (visa32.dll) has a number of entry points that implement Cstyle function calls for doing VISA operations, primarily reading and writing data from/to instruments. While these entry points were designed primarily for use in the C/C++ languages, a number of other languages provide facilities for calling C-style DLL's. One of the most prominent of these languages is Microsoft's Visual Basic 6. The header file for that language, visa32.bas, allows access to most of VISA's functions, but with severe limitations to the formatted I/O functions such as **viPrintf** and **viScanf** due to limitations of the VB 6 interoperability features.

The functions appear in VB .NET as a number of empty functions inside a **Friend Module**. In C# they appear as **static methods** inside an **internal sealed class** that never needs instantiation. They are declared with friend/internal access, so that any libraries that include them will not accidentally export the VISA definitions. The functions are a set of stubs that call out to the VISA C DLL at runtime. At development time, the only thing that the .NET environment knows about them is what is in the header files. There is no equivalent of a .lib file in .NET.

This tutorial demonstrates new, Agilent-defined header files for C# and VB .NET, two new languages that have the ability to call into C DLL's. These new header files do provide some access to the formatted I/O functions, and they can be extended to do anything that could be done in C/C++.

Next: Creating Your Project

creating_your_project

Simply use your development environment, typically Microsoft Visual Studio® .NET, to create a C# or VB .NET project of the type you want (standalone application, library, etc.). Once you have a valid project, you can add Agilent's VISA header file for .NET.

Next: <u>Adding Agilent's VISA Header Files to Your Project</u>

To Link to an Existing Header File

You have two choices for adding the appropriate header file to your project. Depending on how you add the file to your project, you will either link to (reference) the file from wherever it is located on your hard drive, or copy the visa32.bas or visa32.cs file to your project directory. You may choose to link to the file if you plan on modifying it with changes that you want to be used by multiple projects, or if you just don't want extra copies of the file on your system. You may choose to copy the file for a project-specific version if you plan on customizing it for that particular project.

To Link to an Existing Header File

- 1. Right-click the project you wish to modify (not the solution) in the Solution Explorer window of the Microsoft Visual Studio environment.
- 2. Click Add and then click Add Existing Item...
- 3. Navigate to the file you wish to link to in your project (visa32.cs for C# or visa32.vb for Visual Basic .NET), select it, but **do not click the Open button**.
- 4. Click the down arrow to the right of the Open button, and choose Link File.
- 5. You should now see the file underneath your project in the Solution Explorer. It will have a little arrow icon in its lower left corner, indicating that it is a link.

To Make a Project-Specific Copy of the Header File

- 1. Right-click the project you wish to modify (not the solution) in the Solution Explorer window of the Microsoft Visual Studio environment.
- 2. Click Add and then click Add Existing Item...
- 3. Navigate to the file you wish to copy to your project (visa32.cs for C# or visa32.vb for Visual Basic .NET), select it, and **click the Open button**.
- 4. You should now see the file in the Solution Explorer underneath your project. It will have no arrow icon, and you will see a new copy of the file in your project's source file directory.

Next: <u>Using the Header File in Your Project</u>

VB.NET

Now that you have referenced the header file in your project, you can begin development with the VISA functions it contains. This sample demonstrates the simplest VISA transaction.

VB .NET

private void RunSimple()

```
int resourceManager = 0, viError;
int session = 0;
viError = visa32.viOpenDefaultRM(out resourceManager);
viError = visa32.viOpen(resourceManager, "GPIB0::22::INSTR",
visa32.VI_NO_LOCK, visa32.VI_TMO_IMMEDIATE, out session);
viError = visa32.viPrintf(session, "*IDN?\n");
System.Text.StringBuilder idnString =
new System.Text.StringBuilder(1000);
viError = visa32.viScanf(session, "%1000s", idnString);
System.Windows.Forms.MessageBox.Show(idnString.ToString());
visa32.viClose(session);
visa32.viClose(resourceManager);
```

}

This is the "IDN?" query in .NET. You may have noticed the use of the **StringBuilder** class. The **System.String** class in .NET is meant to have an immutable payload, meaning that once you define a string, it is either used or thrown away -- never modified. Since the **viScanf** function modifies the contents of the parameter you pass it, the **System.String** class is illegal for this call. The designers of .NET were aware of this, so they gave the **StringBuilder** class the ability to look like a string buffer when it is passed to C DLL functions.

You need to allocate a sufficiently large buffer for the contents of the read operation, which is why the buffer is allocated 1000-long and why the format string passed to **viScanf**, "%1000s", tells it that there is room in the buffer for 1000 characters. The **MessageBox** knows how much of the buffer to write because **viScanf** appends an ASCII code (0) when it is done reading, and the **StringBuilder** class interprets this as the end of the valid data when you call **StringBuilder.ToString()**.

This solution works perfectly under normal conditions, but what about unexpected errors? What if you're reading in arrays of data? There are some

C#

ł

more techniques to ease debugging and deployment issues with VISA in .NET, as well as some advanced formatted I/O techniques to review.

Next: Adding Error Handling to Your Project

VB.NET

Error handling in .NET is implemented by exceptions. However, you are using the VISA C DLL directly, and it returns error return codes rather than throwing exceptions. One way to handle the errors is to observe the return code of each VISA call, then either return from the method or execute some cleanup code inside the **if** statement that checks the error. Since .NET has excellent support for exceptions, you can use a simple utility function to turn the error codes into exceptions, and wrap your code in a try-catch block.

Below is a utility function for turning VISA errors into

System.ApplicationException exceptions. You may wish to create your own VISA exception class that derives from **System.ApplicationException** in order to better handle VISA errors.

VB .NET

Private Sub CheckStatus(ByVal vi As Integer, ByVal status As Integer) If (status < visa32.VI_SUCCESS) Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(vi, status, err) Throw New ApplicationException(err.ToString()) End If End Sub

This method takes a valid VISA session and a VISA status code as arguments. The session can be either the Resource Manager session or a VISA resource session; the error descriptions returned by **viStatusDesc** will be the same in either case. The method will throw an exception with the error description if the status code is an error code, or do nothing otherwise.

You must wrap any code that calls this helper method in an exception handler, or a VISA error will cause your program to terminate. Here is an example of how to use **CheckStatus**.

VB .NET

```
Dim status As Integer
                         'VISA function status return code
Dim defrm As Integer
                         'Session to Default Resource Manager
Dim vi As Integer
                       'Session to instrument
status = visa32.viOpenDefaultRM(defrm)
' cannot check status on viOpenDefaultRM, since if this failed
' we don't have a valid vi to pass to CheckStatus
Try
  status = visa32.viOpen(defrm, "instrAddress", 0, 0, vi)
  CheckStatus(defrm, status)
  ' ...
Catch err As System.ApplicationException
  MsgBox("*** Error : " & err.Message, vbExclamation,
              "VISA Error Message")
  Exit Sub
End Try
' ...
```

The Try block guarantees that any **ApplicationException** generated by **CheckStatus** will get caught in the Catch block, where an error dialog is displayed.

Next: Reading and Writing Array Data

Fixed-Size Array Length

Often you will need to write or read images, waveforms, or other large, structured data to and from instruments. This data may be in framed binary blocks, ASCII, or other forms. **viScanf** and **viPrintf** can handle the most common instrument formats, and the method overloads provided in the visa32.cs/vb header files provide some of the most common parameter lists.

viPrintf allows writing of array data as comma-separated lists, IEEE 488.2 binary blocks, or raw binary data. There are two ways to tell **viPrintf** the size of the array data: you can hard-code the number into the format string; or you can use the ",*" format flag, meaning that the array size is passed as a 32-bit integer argument in front of the array argument. A call with a hard-coded length might look like this:

Private Function WriteFixedWaveform(ByVal session As Integer, _

ByVal floatData() As Single) As Integer

```
Debug.Assert(floatData.Length >= 100)
```

WriteFixedWaveform = visa32.viPrintf(session, "%,100f" & vbLf, floatData) End Function

The "**,100**" means it is an array of 100 elements to be written as a ASCII comma-separated list. The "**f**" means it contains single-precision floating point numbers. (Note that if your array contains doubles, you must use the **%lf** format specifier.) It is easy to build the format string if you want to pass in an array of arbitrary size and are not concerned about the overhead of some simple string manipulation:

```
Private Function WriteVariableWaveform(ByVal session As Integer, _
ByVal floatData() As Single) As Integer
WriteVariableWaveform = visa32.viPrintf(session, "%," & _
floatData.Length & "f" _
& vbLf, floatData)
```

End Function

See <u>Advanced Use of viPrintf/viScanf</u> for information about declaring more variables to do things like pass an array length argument.

viScanf allows reading of array data in the same formats that **viPrintf** allows for writing. The .NET array is passed by value (meaning that a pointer to the first element is sent) to **viScanf**, which then fills in the array data in the space provided. Because **viScanf** simply writes to the memory location given it, you must verify that the array argument contains enough space to receive all the data. The maximum number of elements to be read can be indicated either through a hard-coded number in the format string or through an extra argument when the "#" format modifier is used. The number of elements actually written is returned in the extra length argument.

Two sets of **viScanf**/**viSScanf** function overloads for arrays are provided in the visa32.vb/cs files: one set with only one array argument, and another with an integer maximum length argument followed by the array argument. Use the one-argument versions when the array size is hard-coded into the format string; use the two-argument version when the "#" format modifier is used. Below are examples of reading arrays using fixed sizes and dynamic sizes respectively.

Fixed-Size Array Length

Private Function ReadKiloBLOB(ByVal session As Integer) As Byte() Dim result() As Byte = New Byte(999) {} Dim statusCode As Long statusCode = visa32.viScanf(session, "%1000y", result) If statusCode < visa32.VI_SUCCESS Then result = Nothing End If Return result End Function

Parameterized-Size Array Length

Private Function ReadFloatList(ByVal session As Integer, ______ ByVal maxSize As Integer) As Single() Dim result() As Single = New Single(maxSize) {} Dim statusCode As Long, elementCount As Long elementCount = maxSize statusCode = visa32.viScanf(session, "%,#f", elementCount, result) If statusCode < visa32.VI_SUCCESS Then result = Nothing ElseIf elementCount <> maxSize Then Dim newArr() As Single = New Single(elementCount) {} Array.Copy(result, newArr, elementCount) result = newArr End If Return result End Function

Next: Putting it All Together

VB.NET

Here is a complete program sample using VISA in VB .NET and in C#. The sample program is written for the Agilent PSA and ESA series spectrum analyzers. It stores the current screen image on the instrument's flash as C:PICTURE.GIF. It then transfers the image over GPIB or LAN and stores the image on your PC in the current directory as picture.gif. The file C:PICTURE.GIF is then deleted from the instrument's flash.

This sample demonstrates reading arrays, error handling, and basic session tasks.

VB .NET

```
Private Sub RunTutorial(ByVal instrAddress As String)
  ' Declare Variables used in the program
  Dim status As Integer
                           'VISA function status return code
  Dim defrm As Integer = 0 'Session to Default Resource Manager
                           'Session to instrument
  Dim vi As Integer = 0
                        'Loop Variable
  Dim x As Integer
  Dim ResultsArray(50000) As Byte 'results array, big enough to hold a GIF
                           'Number of bytes returned from instrument
  Dim length As Integer
  Dim headerlength As Integer 'length of header
  ' the file to write the picture
  Dim fs As System.IO.FileStream = Nothing
  'Set the default number of bytes that will be contained in the
  'ResultsArray to 50,000 (50kB)
  length = 50000
  Try
    If System.IO.File.Exists("picture.gif") Then
       System.IO.File.Delete("picture.gif")
    End If
    ' Open the default resource manager session
    status = visa32.viOpenDefaultRM(defrm)
    ' Open the session. For GPIB, the address string looks like:
         GPIB0::18::INSTR
    ' For PSA, to use LAN, change the string to
    ' "TCPIP0::xxx.xxx.xxx.inst0::INSTR" where
    ' xxxxx is the IP address
    status = visa32.viOpen(defrm, "instrAddress", 0, 0, vi)
    CheckStatus(defrm, status)
    ' Set the I/O timeout to fifteen seconds
    status = visa32.viSetAttribute(vi, visa32.VI_ATTR_TMO_VALUE, 15000)
    CheckStatus(vi, status)
    'Store the current screen image on flash as C:PICTURE.GIF
    status = visa32.viPrintf(vi, ":MMEM:STOR:SCR 'C:PICTURE.GIF'' & vbL
    CheckStatus(vi, status)
    'Grab the screen image file from the instrument
```

```
status = visa32.viPrintf(vi, ":MMEM:DATA? 'C:PICTURE.GIF'' & vbLf)
  CheckStatus(vi, status)
  'We're reading this as raw binary, although it is a IEEE 488.2
  ' binary block containing byte data. We could've used
  ' "%#b" format string, and the byte array would not contain the
  ' IEEE binary block header.
  status = visa32.viScanf(vi, "%#y", length, ResultsArray)
  CheckStatus(vi, status)
  'Delete the tempory file on the flash named C:PICTURE.GIF
  status = visa32.viPrintf(vi, ":MMEM:DEL 'C:PICTURE.GIF'' & vbLf)
  CheckStatus(vi, status)
  'Close the vi session and the resource manager session
  Call visa32.viClose(vi)
  vi = 0
  Call visa32.viClose(defrm)
  defrm = 0
  'Store the results in a text file
  fs = ____
         New System.IO.FileStream("picture.gif", _
         IO.FileMode.OpenOrCreate)
  Dim zeroVal() As Char = {"0"}
  Dim zeroValByte() As Byte
  zeroValByte = System.Text.Encoding.ASCII.GetBytes(zeroVal)
  headerlength = ResultsArray(1) - zeroValByte(0) + 2
  fs.Write(ResultsArray, headerlength, length - 2 - headerlength)
Catch err As System.ApplicationException
  MsgBox("*** Error : " & err.Message, vbExclamation, _
                   "VISA Error Message")
  Exit Sub
Catch err As System.SystemException
  MsgBox("*** Error : " & err.Message, vbExclamation,
                   "System Error Message")
  Exit Sub
Catch err As System. Exception
  Debug.Fail("Unexpected Error")
  MsgBox("*** Error : " & err.Message, vbExclamation, _
                   "Unexpected Error")
```

```
Exit Sub
  Finally
    If Not fs Is Nothing Then fs.Close()
    If vi <> 0 Then
       Call visa32.viClose(vi)
    End If
    If defrm <> 0 Then
       Call visa32.viClose(defrm)
    End If
  End Try
End Sub
Private Sub CheckStatus(ByVal vi As Integer, ByVal status As Integer)
  If (status < visa32.VI_SUCCESS) Then
    Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25
    visa32.viStatusDesc(vi, status, err)
    Throw New ApplicationException(err.ToString())
  End If
End Sub
```

C#

ł

```
private void RunTutorial(string instrAddress)
```

```
// Declare Variables used in the program
              // VISA function status return code
int status:
int defrm = 0; // Session to Default Resource Manager
int vi = 0:
              // Session to instrument
// results array, big enough to hold a GIF
byte[] ResultsArray = new byte[50000];
                   // Number of bytes returned from instrument
int length;
int headerlength;
                   // length of header
System.IO.FileStream fs = null; // the file to write the picture
// Set the default number of bytes that will be contained in the
// ResultsArray to 50,000 (50kB)
length = 50000;
try
{
     if (System.IO.File.Exists("picture.gif"))
          System.IO.File.Delete("picture.gif");
     // Open the default resource manager session
     status = visa32.viOpenDefaultRM(out defrm);
     // Open the session. For GPIB, the address string looks like:
     //
          GPIB0::18::INSTR
     // For PSA, to use LAN, change the string to
     // "TCPIP0::xxx.xxx.xxx.inst0::INSTR" where
     // xxxxx is the IP address
     status = visa32.viOpen(defrm, "instrAddress", 0, 0, out vi);
     CheckStatus(defrm, status);
     // Set the I/O timeout to fifteen seconds
     status = visa32.viSetAttribute(vi,
                   visa32.VI_ATTR_TMO_VALUE, 15000);
     CheckStatus(vi, status);
     //Store the current screen image on flash as C:PICTURE.GIF
     status = visa32.viPrintf(vi,
                    ":MMEM:STOR:SCR 'C:PICTURE.GIF'\n");
```

```
CheckStatus(vi, status);
    //Grab the screen image file from the instrument
    status = visa32.viPrintf(vi, ":MMEM:DATA? 'C:PICTURE.GIF'\n");
    CheckStatus(vi, status);
    // We're reading this as raw binary, although it is a IEEE 488.2
    // binary block containing byte data. We could've used
    // "%#b" format string, and the byte array would not contain the
    // IEEE binary block header.
    status = visa32.viScanf(vi, "%#y", ref length, ResultsArray);
    CheckStatus(vi, status);
    //Delete the tempory file on the flash named C:PICTURE.GIF
    status = visa32.viPrintf(vi, ":MMEM:DEL 'C:PICTURE.GIF'\n");
    CheckStatus(vi, status);
    //Close the vi session and the resource manager session
    visa32.viClose(vi);
    vi = 0;
    visa32.viClose(defrm);
    defrm = 0;
    //Store the results in a text file
    fs = new System.IO.FileStream("picture.gif",
         System.IO.FileMode.OpenOrCreate);
    char[] zeroVal = \{'0'\};
    byte[] zeroValByte;
    zeroValByte = System.Text.Encoding.ASCII.GetBytes(zeroVal);
    headerlength = ResultsArray[1] - zeroValByte[0] + 2;
    fs.Write(ResultsArray, headerlength, length - 2 - headerlength);
catch(System.ApplicationException err)
    System.Windows.Forms.MessageBox.Show("*** Error : " +
         err.Message,
         "VISA Error Message",
         System.Windows.Forms.MessageBoxButtons.OK,
         System.Windows.Forms.MessageBoxIcon.Exclamation);
catch(System.SystemException err)
```

}

{

}

ł

```
System.Windows.Forms.MessageBox.Show("*** Error : " +
              err.Message,
              "System Error Message",
              System.Windows.Forms.MessageBoxButtons.OK,
              System.Windows.Forms.MessageBoxIcon.Exclamation);
    }
    catch(System.Exception err)
    ł
         System.Diagnostics.Debug.Fail("Unexpected Error");
         System.Windows.Forms.MessageBox.Show("*** Error : " +
              err.Message,
              "Unexpected Error",
              System.Windows.Forms.MessageBoxButtons.OK,
              System.Windows.Forms.MessageBoxIcon.Exclamation);
    }
    finally
    {
         if (fs != null)
              fs.Close();
         if (vi != 0)
              visa32.viClose(vi);
         if (defrm != 0)
              visa32.viClose(defrm);
     }
}
private void CheckStatus(int vi, int status)
{
    if (status < visa32.VI_SUCCESS)
    {
         System.Text.StringBuilder err =
                       new System.Text.StringBuilder(256);
         visa32.viStatusDesc(vi, status, err);
         throw new ApplicationException(err.ToString());
    }
}
```

Next: Deploying Your Project
deploying_your_project

The only system requirements (specific to VISA) for deploying your compiled programs on other machines are:

- A valid VISA32.DLL must be in the system's **PATH** environment variable.
- The resource address you are trying to open must exist on the system and be configured for the VISA32.DLL that is found first during the Windows DLL search.

These requirements refer only to what VISA requires to work. You will still have to satisfy the normal .NET requirements, such as having the .NET framework installed on the deployed systems. Obviously, any other software libraries your program uses at runtime must be installed as well.

Because each VISA vendor installs its version of the VISA DLL, you may have a different VISA DLL being used on your deployed system than the one with which you developed your application. When multiple vendors' VISA implementations are present, the DLL used is the one that is found first using Microsoft Windows' DLL search rules. If you developed your program using Agilent VISA, and you wish your program to use Agilent VISA no matter what other VISA implementations are on your deployed systems, you can change the DLL name in all of the method declarations in visa32.cs or visa32.vb from "VISA32.DLL" to "AGVISA32.DLL." This will prevent your program from working with any other company's VISA implementation, and will direct your program to use the Agilent DLL if multiple VISA DLL's are installed on the system.

C#

The **viPrintf** and **viScanf** VISA functions take variable argument lists in C. Some of viScanf's variable arguments are references to primitive types (int, float, etc), meaning that the values themselves can be changed by the function. There is no equivalent in either C# or VB .NET for variable argument lists with reference arguments, so no direct translation is available. There are undocumented C# keywords (**__arglist** and **__makeref**, primarily) for calling printf- and scanf-style variable argument lists, but there is no guarantee of future support and these keywords do not work in VB .NET.

Agilent has chosen to implement a set of basic one-argument method overloads for **viScanf**, **viPrintf**, **viSScanf**, and **viSPrintf**. This allows you to pass one argument to the functions, and the proper overload will be chosen based on the type of the argument. Overloads for **viScanf/viSScanf** for reading arrays with the argument length parameter are also provided. There are a total of 58 overloads in each header file to support this. Adding a second parameter for all the functions and providing all the overloads would result in 484 overloads in each file, which is infeasible. If you find yourself needing more choices than are in the Agilent-provided header files, you can make your own declarations based on your needs.

Let's look at one of the overloads provided in the C# header file:

```
[DllImportAttribute("VISA32.DLL", EntryPoint="#269",
ExactSpelling=true, CharSet=CharSet.Ansi,
SetLastError=true, CallingConvention=CallingConvention.Cdecl)]
public static extern int viPrintf(int vi, string writeFmt, int[] arr);
```

This declaration allows **viPrintf** to write out an array of integers. If, for example, you wanted to upload a waveform to an arbitrary waveform generator, you would typically send a SCPI header followed by the array data. **viPrintf** doesn't understand .NET arrays, so it isn't aware of the size of the array; it only has a pointer to data. The format string has two ways to inform **viPrintf** of the size of the array: a number can be in the format string itself, or the "*" symbol can be used to indicate that a parameter indicating the array size is passed as the first argument.

The provided overload allows you to hard-code the format string with that SCPI header and the size of the array. You could create a format string on the fly with the proper array size, but you might choose to avoid the overhead of string manipulation to create the format string. Another solution that reduces your complexity and amount of code is to declare another override of **viPrintf** that allows for that extra parameter. If this is something you want to do, you might declare your own version of **viScanf** like this:

C#

VB .NET

<DllImportAttribute("VISA32.DLL", EntryPoint:="#269", ExactSpelling:=True, CharSet:=CharSet.Ansi, SetLastError:=True, _ CallingConvention:=CallingConvention.Cdecl)> _ Public Function viPrintf(ByVal vi As Integer, ByVal writeFmt As String, _ ByVal pointCount As Integer, _ ByVal waveform() As Short) As Integer

End Function

You might use this version in a function like this:

VB .NET

```
Private Sub SendWaveform(ByVal session As Integer, ByVal pointData() As Sho
Dim statusCode As Integer
statusCode = viPrintf(session, "DATA:DAC VOLATILE, %*hb" & vbLf, _
pointData.Length(), pointData)
If (statusCode < visa32.VI_SUCCESS) Then
Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25
visa32.viStatusDesc(session, statusCode, err)
Throw New ApplicationException(err.ToString())
End If
End Sub
```

The "%,*hb" format string means, in order, a parameter (%), that is an array with the length of the array data passed as another parameter (*), that is passed on the stack as a reference to an array of 16-bit integers (h), to be written to the instrument as an IEEE 488.2 binary block (b). You can create these new function declarations because .NET simply sends the declared parameters to the underlying VISA C functions, and C variable argument list functions know how to parse the format string so that they can use the stack to get at the right data.

Safety

As in the C/C++ world, you have to be very careful when using these methods. Even though .NET protects its data internally, when it calls out to C DLL's it is up to the DLL to behave properly. If the type that you pass to **viPrintf/viScanf** is different from the type the format string promises, you will have undefined behavior and possible program crashes. For example, if you call **viScanf** with an 10-long array of integers, but the format string is "%dl,100" (promising space to write 100 integers), you can cause an illegal write to the program heap. This may result in the overwriting of program data and/or an illegal memory access exception.

dotnet_types_and_visa_c_types_reference

VISA Type	.NET Type	Description
ViInt8, ViUInt8, ViChar, ViByte	System.Byte	Since signed bytes are not <u>CLS¹</u> -compliant, we use the byte type for both.
ViInt16, ViUInt16	System.Int16	Since unsigned shorts are not CLS-compliant, we use the signed type for both.
ViInt32, ViUInt32	System.Int32	Since unsigned integers are not CLS- compliant, we use the signed type for both.
ViSession, ViPFindList, ViPEvent	System.Int32	Sessions and other VISA object handles are just 32-bit integers.
ViStatus	System.Int32	Status codes (error codes) have always been 32-bit integers.
ViAddr	System.Int32	This is a reference to a remote 32-bit memory space, and will therefore fit in a 32-bit integer
ViBuf	System.String, System.Byte[]	This type represents byte buffers or ASCII strings that are not to be modified by VISA. The System.String class was chosen for methods that typically write ANSI strings. .NET automatically marshals the System.String class into ASCII strings that VISA can accept. For methods that are typically binary data, a byte array was chosen. It is marshaled such that VISA gets a pointer to the first element of the array.

ViPBuf	System.StringBuilder, System.Byte[]	This type represents byte buffers or ASCII string buffers that are to be written to by VISA. System.StringBuilder was chosen for functions that typically write ASCII strings, because the default marshalling behavior of that class is to give the C function a pointer to the beginning of the preallocated buffer, and to use the 0 ASCII value to determine the end of the string written in the buffer. Don't forget to allocate enough storage space in your StringBuilder object. System.Byte was chosen for functions that often return binary data. It is marshaled such that VISA gets a pointer to the first element of the array.
ViChar[]	System.StringBuilder	Arrays of ViChar are used when VISA plans to write short-length strings into the passed-in buffer. Again, System.Stringbuilder is well- suited to acting as a character buffer.
ViAttrState	System.Int32, System.Int16, System.Byte	The value passed to viSetAttribute is an 8-, 16-, or 32-bit integer.
void *	C#: out byte, out short, out int, StringBuilder VB .NET: ByRef Byte, ByRef Short, ByRef Integer, StringBuilder	viSetAttribute can return 8-, 16-, or 32-bit integers, or strings.
		The ViPXXXX types, where 'XXXXX' is an integral type, are return values for integer types. In C#, the 'out' keyword means that the value going in doesn't matter and only the

ViPUInt8, ViPInt8, etc.	C#: out byte, etc VB .NET: ByRef Byte, etc	return value matters. VB .NET does not have the 'out' keyword, so a reference is indicated by using the 'ByRef' keyword. Therefore, you must initialize values in VB .NET even though they will be overwritten. The .NET marshalling behavior of 'out' and 'ByRef' is to pass the argument as pointer to the value, which is what VISA is expecting for the ViPXXXX types.

¹Common Language Specification. The specification of the set of types and .NET constructs that are applicable across VB .NET, C#, and all other .NET languages. API designers or others concerned with compatibility with multiple .NET languages typically limit the types they use to CLS-compliant types. See the <u>Microsoft Developer Network</u> .NET documentation for more details.

Documentation Notes

About This Help

Revised: September 2012

Current with Agilent IO Libraries Suite revision 16.3

Copyright © 1995-1996, 1998, 2000-2012 Agilent Technologies, Inc.

Warranty

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

Technology Licenses

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

Restricted Rights Legend

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

Trademark Information

Microsoft® is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

PCIeTM and PCI EXPRESS® are registered trademarks and/or service marks of PCI-SIG.

Visual Studio® is a trademark or registered trademark of Microsoft Corporation in the United States and/or other countries.

Windows® and MS Windows® are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Windows Vista® is a trademark or registered trademark of Microsoft Corporation in the United States and/or other countries.

Reproduced with Permission

Content from the IVI specifications reproduced with permission from the <u>IVI</u> <u>Foundation</u>.

The IVI Foundation and its member companies make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The IVI Foundation and its member companies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Glossary

address alias API attribute bus error bus error handler <u>commander</u> communication channel **Connection Expert** Controller device device driver device session driver explorer view handler **HiSLIP** instrument instrument driver **Interactive IO** interface interface driver interface session <u>interrupt</u> **IO Control IO** Libraries lock mapping non-Controller role notification area operation

primary VISA process **PXI** <u>refresh</u> register resource (or resource instance) resource class resource descriptor **SCPI** secondary VISA session **SICL** side-by-side <u>SRQ</u> status byte symbolic name task guide taskbar notification area test system thread ViFind32 virtual instrument **VISA VISA address VISA** alias **VISA COM** VISA Instrument Control Resources **VISA** name VISA resource manager VISA resource template VXI Resource Manager Windows notification area

auto-discovery_address_check_and_vifindrsrc

If you are a VISA programmer, or you use applications developed with the VISA I/O library, you use the **viFindRsrc** and **viFindNext** functions to discover and list instruments in your test system. This topic explains the interactions between Agilent Connection Expert settings and **viFindRsrc/viFindNext**.

GPIB and LAN instruments have an **address-check** property, which can be set in Connection Expert. The scanning algorithm employed by **viFindRsrc/viFindNext** uses the address-check property to determine whether to scan for the instrument or to assume its presence. In addition, **viOpen** uses this property to decide whether to verify the presence of the instrument. Therefore,

- When address-check is on:
 - viFindRsrc/viFindNext will scan for an instrument at the given address. If the instrument is present and turned on, it will be discovered and returned by viFindRsrc/viFindNext. If the instrument is missing or nonfunctional (e.g., turned off), it will not be returned.
 - **viOpen** will verify the presence of the instrument on which you are trying to open a session.
- When address-check is off:
 - **viFindRsrc/viFindNext** assume the presence of an instrument at that address. The instrument will be included in the returned values from **viFindRsrc/viFindNext** regardless of whether the instrument is actually present.
 - **viOpen** will not verify the presence of the instrument. If the device is not present or not turned on, you will not discover this until you attempt to communicate with the instrument. **viOpen** will take less time to execute if address-check is off.

viVPrintf/viVScanf Example using String

Sub Test_String()

' Using viVScanf and viVPrintf in Visual Basic 6 is tricky because ' of the way VB6 passes arguments on the stack. The 'vararg' ' argument to the data needs to be a 'byRef' pointer -- which means ' it needs to be a pointer to a pointer.

' VB treats string arrays passed to external DLL's as a special case ' so when a string is passed, it ends up being passed correctly as a ' pointer to a pointer.

' Numeric arrays, on the other hand, cannot be passed directly. To ' pass a pointer to the numeric array pointer, we create another ' variable which we set to the address of the start of the array ' and pass that variable 'byRef' to the function.

```
' Declare variables used by VISA
```

Dim err As Long Dim drm As Long Dim vi As Long Dim buf As String

' Open a default resource manager session and open the device

```
err = viOpenDefaultRM(drm)
err = viOpen(drm, "GPIB2::2::INSTR", 0, 0, vi)
```

' Read a string array using viVScanf

Dim buf256 As String * 256 ' pre-allocated, string Dim posn As Long

```
' Tell the instrument to send its IDN string
err = viVPrintf(vi, "*IDN?" + vbLf, 0)
' Read the string from the instrument
err = viVScanf(vi, "%t", buf256)
' Truncate the returned string at the first null
' (A null terminated string is returned)
posn = InStr(buf256, String(1, 0))
If (posn > 0) Then
buf = Left(buf256, posn - 1)
Else
buf = buf256
End If
' display the string that was read from the instrument
MsgBox buf, 0, "viVScanf of a String"
' Close the VISA sessions
err = viClose(vi)
err = viClose(drm)
End Sub
```

See Also

viVPrintf, viVScanf Using the VISA C API in Microsoft Visual Basic 6

viVQueryf Example with String and Indefinite Length Block

Sub Test_viVQueryf()

' Using viVScanf and viVPrintf in Visual Basic 6 is tricky because ' of the way VB6 passes arguments on the stack. The 'vararg' ' argument to the data needs to be a 'byRef' pointer -- which means ' it needs to be a pointer to a pointer.

' VB treats string arrays passed to external DLL's as a special case ' so when a string is passed, it ends up being passed correctly as a ' pointer to a pointer.

' Numeric arrays, on the other hand, cannot be passed directly. To ' pass a pointer to the numeric array pointer, we create another ' variable which we set to the address of the start of the array ' and pass that variable 'byRef' to the function.

```
' Declare variables used by VISA
```

Dim err As Long Dim drm As Long Dim vi As Long Dim buf As String

' Open a default resource manager session and open the device

```
err = viOpenDefaultRM(drm)
err = viOpen(drm, "GPIB2::2::INSTR", 0, 0, vi)
```

' _____

'Write a String and read an indefinite length block

' Set up the test instrument to send an IEEE-488

' indefinite length block terminated with a line-feed.

```
err = viVPrintf(vi, "RECEIVE" & vbLf, 0)
err = viVPrintf(vi, "#0abcdefghij" & vbLf, 0)
' This now done in viVQueryf:
   ----> err = viVPrintf(vi, "SEND" & vbLf, 0)
' Declare the byte array, a count variable and a
' array to hold the parameter addresses for viVScanf
Dim retCount As Long
Dim byteArray(20) As Byte
Dim paramsArray(3) As Long
' Create and initialize a string for the 'writeFmt'
' parameter of viVQueryf. Note that we can't put
' pointers to strings in the paramsArray because
' strings in VB are unicode and viVQueryf needs
' a ascii strings.
Dim writeFmtString As String
writeFmtString = "SEND"
' Put the parameter addresses in the array
paramsArray(0) = VarPtr(retCount)
paramsArray(1) = VarPtr(byteArray(0))
' Set 'retCount to the maximum number of elements
' that the array can hold.
retCount = UBound(byteArray) - LBound(byteArray) + 1
' Read the array from the test instrument and set
' the retCount variable to the actual count of
' elements read. Note that a line-feed is appended
' to the format strings. On the writeFmt string,
' this is causes a flush of the formatted write buffer
```

' to the device. The line-feed must be in the format

' and not simply in the data since a line-feed in the

' data will not cause a flush. On the readFmt string,

' the line-feed ensures that we flush the new-line out

' of the read buffer and don't leave it for a future

' formatted read to trip over.

```
err = viVQueryf(vi, writeFmtString & vbLf, "%#b" & vbLf, paramsArray(0))
```

' Display some of the results

See Also

viVQueryf Using the VISA C API in Microsoft Visual Basic 6

viVScanf Example Returning a Double Array

Sub Test_Double()

' Using viVScanf and viVPrintf in Visual Basic 6 is tricky because ' of the way VB6 passes arguments on the stack. The 'vararg' ' argument to the data needs to be a 'byRef' pointer -- which means ' it needs to be a pointer to a pointer.

' VB treats string arrays passed to external DLL's as a special case ' so when a string is passed, it ends up being passed correctly as a ' pointer to a pointer.

' Numeric arrays, on the other hand, cannot be passed directly. To ' pass a pointer to the numeric array pointer, we create another ' variable which we set to the address of the start of the array ' and pass that variable 'byRef' to the function.

```
' Declare variables used by VISA
```

Dim err As Long Dim drm As Long Dim vi As Long Dim buf As String

' Open a default resource manager session and open the device

```
err = viOpenDefaultRM(drm)
err = viOpen(drm, "GPIB2::2::INSTR", 0, 0, vi)
```

```
.
```

' Read a double array

' Set up the test instrument to send a comma

' separated array of floating point values

' terminated with a line-feed.

```
1
err = viVPrintf(vi, "RECEIVE" & vbLf, 0)
err = viVPrintf(vi, "1.1,2.2,3.3,4.4,5.5" & vbLf, 0)
err = viVPrintf(vi, "SEND" & vbLf, 0)
' Declare the numeric array, a count variable and a
' array to hold the parameter addresses for viVScanf
Dim retCount As Long
Dim dblArray(20) As Double
Dim paramsArray(2) As Long
' Put the parameter addresses in the array
paramsArray(0) = VarPtr(retCount)
paramsArray(1) = VarPtr(dblArray(0))
' Set 'retCount to the maximum number of elements
' that the array can hold.
retCount = UBound(dblArray) - LBound(dblArray) + 1
' Read the array from the test instrument and set
' the retCount variable to the actual count of
' elements read. Note that the format string
' specifies the line-feed -- this is needed to
' sure the line-feed is flushed from the formatted
' IO buffer.
err = viVScanf(vi, "%,#lf" & vbLf, paramsArray(0))
' Display some of the results
buf = Format(err, "Error = ##########0") & vbLf &
Format(retCount, "Retur\n \Cou\nt = #########0") & vbLf &
Format(dblArray(0), "##0.0, ") & vbLf & _
Format(dblArray(1), "##0.0, ") & vbLf & _
```

Format(dblArray(2), "##0.0, ") & vbLf & _ Format(dblArray(3), "##0.0, ") & vbLf & _ Format(dblArray(4), "##0.0, ") & vbLf MsgBox buf, 0, "viVScanf of a Double Array"

' Close the VISA sessions

err = viClose(vi) err = viClose(drm) End Sub See Also

<u>viVScanf</u> <u>Using the VISA C API in Microsoft Visual Basic 6</u>

viVScanf Example Reading an IEEE 488 Definite Length Block and Returning a Byte Array

```
Sub Test_DefiniteLengthBlock()
' Using viVScanf and viVPrintf in Visual Basic 6 is tricky because
' of the way VB6 passes arguments on the stack. The 'vararg'
' argument to the data needs to be a 'byRef' pointer -- which means
' it needs to be a pointer to a pointer.
'
' VB treats string arrays passed to external DLL's as a special case
```

' so when a string is passed, it ends up being passed correctly as a ' pointer to a pointer.

' Numeric arrays, on the other hand, cannot be passed directly. To ' pass a pointer to the numeric array pointer, we create another ' variable which we set to the address of the start of the array ' and pass that variable 'byRef' to the function.

```
' Declare variables used by VISA
```

Dim err As Long Dim drm As Long Dim vi As Long Dim buf As String

' Open a default resource manager session and open the device

```
err = viOpenDefaultRM(drm)
err = viOpen(drm, "GPIB2::2::INSTR", 0, 0, vi)
```

' Read an IEEE 488 definite length block

' Set up the test instrument to send an IEEE-488

' definite length block terminated with a line-feed.

```
err = viVPrintf(vi, "RECEIVE" & vbLf, 0)
err = viVPrintf(vi, "#210abcdefghij" & vbLf, 0)
err = viVPrintf(vi, "SEND" & vbLf, 0)
' Declare the byte array, a count variable and a
' array to hold the parameter addresses for viVScanf
Dim retCount As Long
Dim byteArray(20) As Byte
Dim paramsArray(2) As Long
' Put the parameter addresses in the array
paramsArray(0) = VarPtr(retCount)
paramsArray(1) = VarPtr(byteArray(0))
' Set 'retCount to the maximum number of elements
' that the array can hold.
retCount = UBound(byteArray) - LBound(byteArray) + 1
' Read the array from the test instrument and set
' the retCount variable to the actual count of
' elements read. Note that the format string
' specifies the line-feed -- this is needed to
' sure the line-feed is flushed from the formatted
' IO buffer.
err = viVScanf(vi, "%#b" & vbLf, paramsArray(0))
' Display some of the results
buf = Format(err, "Error = ##########0") & vbLf &
Format(retCount, "Retur\n \Cou\nt = #########0") & vbLf &
Chr(byteArray(0)) & Chr(byteArray(1)) & Chr(byteArray(2)) & _
Chr(byteArray(3)) & Chr(byteArray(4)) & Chr(byteArray(5)) & _
```

Chr(byteArray(6)) & Chr(byteArray(7)) & Chr(byteArray(8)) & _ Chr(byteArray(9)) & vbLf MsgBox buf, 0, "viVScanf of an IEEE 488 Definite Length Byte Array"

' Close the VISA sessions

err = viClose(vi) err = viClose(drm) End Sub See Also

viVScanf Using the VISA C API in Microsoft Visual Basic 6

viVScanf Example Reading an IEEE 488 Indefinite Length Block and Returning a Byte Array

Sub Test_IndefiniteLengthBlock()
' Using viVScanf and viVPrintf in Visual Basic 6 is tricky because
' of the way VB6 passes arguments on the stack. The 'vararg'
' argument to the data needs to be a 'byRef' pointer -- which means
' it needs to be a pointer to a pointer.
' VB treats string arrays passed to external DLL's as a special case
' so when a string is passed, it ends up being passed correctly as a

' pointer to a pointer.

' Numeric arrays, on the other hand, cannot be passed directly. To ' pass a pointer to the numeric array pointer, we create another ' variable which we set to the address of the start of the array ' and pass that variable 'byRef' to the function.

```
' Declare variables used by VISA
```

Dim err As Long Dim drm As Long Dim vi As Long Dim buf As String

' Open a default resource manager session and open the device

```
err = viOpenDefaultRM(drm)
err = viOpen(drm, "GPIB2::2::INSTR", 0, 0, vi)
```

' Read an IEEE 488 indefinite length block

' Set up the test instrument to send an IEEE-488

' indefinite length block terminated with a line-feed.
```
err = viVPrintf(vi, "RECEIVE" & vbLf, 0)
err = viVPrintf(vi, "#0abcdefghij" & vbLf, 0)
err = viVPrintf(vi, "SEND" & vbLf, 0)
' Declare the byte array, a count variable and a
' array to hold the parameter addresses for viVScanf
Dim retCount As Long
Dim byteArray(20) As Byte
Dim paramsArray(2) As Long
' Put the parameter addresses in the array
paramsArray(0) = VarPtr(retCount)
paramsArray(1) = VarPtr(byteArray(0))
' Set 'retCount to the maximum number of elements
' that the array can hold.
retCount = UBound(byteArray) - LBound(byteArray) + 1
' Read the array from the test instrument and set
' the retCount variable to the actual count of
' elements read. Note that the format string
' specifies the line-feed -- this is needed to
' sure the line-feed is flushed from the formatted
' IO buffer.
err = viVScanf(vi, "%#b" & vbLf, paramsArray(0))
' Display some of the results
buf = Format(err, "Error = ##########0") & vbLf &
Format(retCount, "Retur\n \Cou\nt = #########0") & vbLf &
Chr(byteArray(0)) & Chr(byteArray(1)) & Chr(byteArray(2)) & _
Chr(byteArray(3)) & Chr(byteArray(4)) & Chr(byteArray(5)) & _
```

Chr(byteArray(6)) & Chr(byteArray(7)) & Chr(byteArray(8)) & _ Chr(byteArray(9)) & vbLf MsgBox buf, 0, "viVScanf of an IEEE 488 Indefinite Length Byte Array"

' Close the VISA sessions

err = viClose(vi) err = viClose(drm) End Sub See Also

viVScanf Using the VISA C API in Microsoft Visual Basic 6

00

Syntax

viOpenDefaultRM(out int sesn);

Description

Opens the default VISA Resource Manager, the <u>Class Factory</u>¹ VISA Object that knows how to find and open VISA resources. This function must be called before any other 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.

Parameters

Name	<u>Dir</u>	Туре	Description
sesn	OUT	int	Unique logical identifier to a Default Resource Manager session.

Return Values

Completion Code	Description	
VI_SUCCESS	Session to the Default Resource Manager resource created successfully.	
Error Codes	Description	
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.	
VI_ERROR_SYSTEM_ERROR	The VISA system failed to initialize.	

C# Example

VB.NET Example

Public Function OpenRM() As Integer Dim resourceManager As Integer = 0, viError As Integer viError = visa32.viOpenDefaultRM(resourceManager) If viError < visa32.VI_SUCCESS Then Throw New ApplicationException("Failed to open Resource Manager") End If Return resourceManager End Function

¹See the excellent book *Design Patterns*, Gamma et al, for more information about the Class Factory pattern and other commonly used patterns in software systems.

Syntax

Syntax

```
viScanf(int vi, string readFmt, <overloaded
    parameter>);
```

Description

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

For information about getting more options in the parameter list, see <u>Advanced</u> <u>Use of viPrintf/viScanf</u>.

Parameters

Name	<u>Dir</u>	Туре	Description
vi	IN	int	Unique logical identifier to a session.
readFmt	IN	string	String describing the format for arguments.
overloaded argument	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

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

C# Example

VB.NET Example

Public Function ReadDefiniteLengthBinaryBlockAsInt32(ByVal session As Inte ByVal maxElementCount As Integer, _ ByVal data() As Integer) As Integer Dim viError As Integer, elementCount As Integer = maxElementCount viError = visa32.viScanf(session, "%#lb", elementCount, data) If viError < visa32.VI_SUCCESS Then Dim err As System.Text.StringBuilder = New System.Text.StringBuilder(25 visa32.viStatusDesc(session, viError, err) Throw New ApplicationException(err.ToString()) End If Return elementCount End Function