HEWLETT PACKARD

User's Guide

HP VISA

HP VISA

User's Guide

Notice

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

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

Warranty Information.

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend.

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

Microsoft, Windows NT, and Windows 95 are U.S. registered trademark of Microsoft Corporation.

Pentium is a U.S. registered trademark of Intel Corporation.

UNIX is a registered trademark of the United States and other countries, licensed exclusively through X/Open Company Limited.

Printing History

This is the second edition of the *HP VISA User's Guide.*

May 1996 — First Edition

September 1996 — Second Edition

Contents

1. Introduction

HP VISA Overview 1-4
Windows Support 1-4
HP-UX Support 1-5
Users 1-6
Other Documentation 1-7
Where to Go Next 1-8

2. Building an HP VISA Application in Windows

Reviewing an HP VISA Program 2-3
Example Program Contents 2-5
`visa.h` 2-5
`ViSession` 2-5
`viOpenDefaultRM` 2-5
`viOpen` 2-5
`viPrintf` and `viScanf` 2-5
`viClose` 2-5
Linking to HP VISA Libraries 2-6
Compiling and Linking an HP VISA Program 2-7
32-bit Applications 2-7
16-bit Applications 2-8
Logging Error Messages 2-11
Windows 95 2-11
Windows NT 2-11
Running an HP VISA Program 2-12
Where to Go Next 2-13

3. Building an HP VISA Application in HP-UX

Reviewing an HP VISA Program 3-3
The Example Program Contents 3-5
`visa.h` 3-5
`ViSession` 3-5
`viOpenDefaultRM` 3-5
`viOpen` 3-5
`viPrintf` and `viScanf` 3-5
`viClose` 3-5

Compiling and Linking an HP VISA Program 3-6
Logging Error Messages 3-7
Running an HP VISA Program 3-8
Getting Online Help 3-9
Using the HyperHelp Viewer 3-9
Using HP-UX Manual Pages 3-10
Where to Go Next 3-11

4. Programming with HP VISA
Including the HP VISA Declarations File 4-3
Opening a Session 4-4
Device Sessions 4-5
Addressing a Session 4-7
Closing a Session 4-9
Searching for Resources 4-10
Sending I/O Commands 4-13
Formatted I/O 4-14
Formatted I/O Conversion 4-15
Modifiers 4-15
Field Width 4-15
. Precision 4-16
Argument Length Modifier 4-17
, Array Size 4-18
Special Characters 4-18
Conversion Characters 4-19
Formatted I/O Example 4-20
Format String 4-22
Formatted I/O Buffers 4-22
Non-Formatted I/O 4-23
Non-Formatted I/O Example 4-24
Using Attributes 4-26
HP VISA Resource Attributes 4-27
HP VISA Generic Instrument Attributes 4-28
HP VISA Interface Specific Instrument Attributes . . . 4-29
GPIB and GPIB-VXI Interfaces 4-29
VXI and GPIB-VXI Interfaces 4-29
GPIB-VXI Interface 4-31
ASRL Interface 4-31
HP VISA Event Attributes 4-32
Using Events and Handlers 4-33
Events and Attributes 4-35

Reading the Attribute 4-36
The Callback Method 4-37
Installing Handlers 4-37
Writing the Handler 4-38
Enabling Events 4-39
Event Callback Example 4-40
SRQ Callback Example 4-42
The Queuing Method 4-45
Enabling Events 4-45
Wait on the Event 4-46
Event Queuing Example 4-47
Trapping Errors 4-49
HP VISA Errors 4-49
Instrument Errors 4-50
Using Locks 4-51
Lock Types 4-53
Lock Sharing 4-54
Acquiring an Exclusive Lock While Holding a Shared Lock 4-55
Nested Locks 4-56
Lock Examples 4-56

5. Programming VXI Devices

Programming Overview 5-3
Using High-Level Memory Functions 5-5
Programming to the Registers 5-6
High-Level Memory Functions Examples 5-9
Using Low-Level Memory Functions 5-11
Programming to the Registers 5-11
Mapping Memory Space 5-12
Reading and Writing to the Device Registers 5-13
Unmapping Memory Space 5-13
Low-Level Memory Functions Examples 5-14
Considering VXI Backplane Memory I/O Performance . . 5-17
Using VXI Specific Attributes 5-23
Using the Map Address as a Pointer 5-23
Setting the VXI Trigger Line 5-25

6. Programming over LAN

Overview of the LAN 6-4
LAN Software Architecture 6-6
LAN Networking Protocols 6-7
LAN Client and Threads 6-8
LAN Server 6-8
Considering LAN Configuration and Performance 6-9
Communicating with Devices over LAN 6-10
Addressing a Session 6-10
LAN Session Example 6-11
Using Timeouts with LAN 6-13
Default LAN Timeout Values 6-14
Application Terminations and Timeouts 6-16
Using Signal Handling with LAN 6-17
SIGIO Signals 6-17
HP VISA Function Support with LAN 6-18
GPIB Sessions and Service Requests over LAN 6-18

7. HP VISA Language Reference

viAssertTrigger 7-7
viClear 7-9
viClose 7-11
viDisableEvent 7-13
viDiscardEvents 7-16
viEnableEvent 7-18
viEventHandler 7-21
viFindNext 7-24
viFindRsrc 7-25
viFlush 7-27
viGetAttribute 7-30
viIn8, viIn16, and viIn32 7-32
viInstallHandler 7-34
viLock 7-36
viMapAddress 7-41
viMemAlloc 7-44
viMemFree 7-46
viMoveIn8, viMoveIn16, and viMoveIn32 7-47
viMoveOut8, viMoveOut16, and viMoveOut32 7-50
viOpen 7-53
viOpenDefaultRM 7-55
viOut8, viOut16, and viOut32 7-57

viPeek8, viPeek16, and viPeek32 7-59
viPoke8, viPoke16, and viPoke32 7-61
viPrintf 7-63
viQueryf 7-71
viRead 7-73
viReadAsync 7-76
viReadSTB 7-78
viScanf 7-80
viSetAttribute 7-87
viSetBuf 7-89
viStatusDesc 7-91
viTerminate 7-92
viUninstallHandler 7-93
viUnlock 7-95
viUnmapAddress 7-97
viVPrintf 7-98
viVQueryf 7-100
viVScanf 7-102
viWaitOnEvent 7-104
viWrite 7-107
viWriteAsync 7-109

A. HP VISA System Information

Windows Directory Structure A-3
UNIX Directory Structure A-5
About the Directories A-6
The `HPVISA` Subdirectory A-6
Include Files A-6
Libraries A-6
Sample Programs A-7
VXI*plug&play* Instrument Drivers A-7

B. HP VISA Attributes

HP VISA Resource Attributes B-3
HP VISA Generic Instrument Attributes B-4
HP VISA Interface Specific Instrument Attributes B-5
GPIB and GPIB-VXI Interfaces B-5
VXI and GPIB-VXI Interfaces B-6
GPIB-VXI Interface B-7
ASRL Interface B-8
HP VISA Event Attributes B-9

C. HP VISA Completion and Error Codes
Alphabetized Completion and Error Codes C-3
Completion and Error Codes for Each HP VISA Function C-7

D. HP VISA Type Definitions

E. Editing the HP VISA Configuration
On Windows 95 and Windows NT E-3
On HP-UX E-5

Glossary

Index

1

Introduction

Introduction

Welcome to the *HP VISA User's Guide*. This manual describes the HP VISA (Virtual Instrument Software Architecture) library and how to use it to develop instrument drivers and I/O applications on Microsoft® Windows 95® and Windows NT®, as well as on HP-UX version 10.20 or later.

Before using VISA, you must install and configure VISA according to the instructions in the *HP I/O Libraries Installation and Configuration Guide*.

This first chapter provides an overview of VISA. In addition, this guide contains the following chapters:

- **Chapter 2 - Building an HP VISA Application in Windows** describes how to build a VISA application in a Microsoft Windows environment. A simple example program is also provided to help you get started programming with VISA.
- **Chapter 3 - Building an HP VISA Application in HP-UX** describes how to build a VISA application in the HP-UX environment. A simple example program is also provided to help you get started programming with VISA.
- **Chapter 4 - Programming with HP VISA** describes the basics of VISA, along with some detailed example programs. You can find information on creating sessions, and on using formatted I/O, events and handlers, attributes, locking, and more.
- **Chapter 5 - Programming VXI Devices** describes how to use VISA to communicate over the VXI and GPIB-VXI interfaces to VXI instruments.
- **Chapter 6 - Programming over LAN** provides an overview of the LAN and describes how to use VISA to communicate with devices over LAN.
- **Chapter 7 - HP VISA Language Reference** describes the supported VISA functions. These functions are provided in alphabetical order to make them easy to look-up and reference.

This guide also contains the following appendices:

- **Appendix A - HP VISA System Information** provides information on VISA software files and system interaction.
- **Appendix B - HP VISA Attributes** provides a table of all VISA attributes and their associated values.
- **Appendix C - HP VISA Completion and Error Codes** lists all the completion and error codes for VISA.
- **Appendix D - HP VISA Type Definitions** lists the VISA data types and their definitions.
- **Appendix E - Editing the HP VISA Configuration** describes how to edit the VISA configuration to gain better performance.

This guide also includes a **Glossary** of terms and their definitions, as well as an **Index**.

HP VISA Overview

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

Windows Support

There is a 32-bit version of VISA on both Windows 95 and Windows NT, and a 16-bit version of VISA on Windows 95. Note that you can use one or both versions of VISA (32-bit and/or 16-bit VISA) on your 32-bit computer when running Windows 95.

The following two tables summarize the support for the 32-bit and 16-bit versions of VISA on Windows environments.

Support for 32-bit VISA on Windows 95 and Windows NT

Interfaces	Programming Languages
GPIB, VXI[1], GPIB-VXI, RS-232, LAN[2]	C, C++, Visual BASIC[3]

[1] VISA for the VXI interface on Windows NT (version 4.0 or later) is shipped with the HP VXI Pentium® Controller product only.

[2] LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

[3] Although VISA for Windows supports the Visual BASIC programming language, this manual only supports and shows VISA programming techniques using the C and C++ programming languages at this time.

Support for 16-bit VISA on Windows 95

Interfaces	Programming Languages
GPIB, VXI, GPIB-VXI, RS-232	C, C++, Visual BASIC[4]

[4] Although VISA for Windows supports the Visual BASIC programming language, this manual only supports and shows VISA programming techniques using the C and C++ programming languages at this time.

HP-UX Support

The following table summarizes the support for VISA on HP-UX version 10.20 or later.

Support for VISA on HP-UX Version 10.20 or Later

Interfaces	Programming Languages
GPIB, VXI, GPIB-VXI, LAN[5]	C, C++

[5] LAN support from within VISA occurs via an address translation such that a GPIB interface can be accessed remotely over a computer network.

Users

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

Software development using VISA is intended for instrument I/O programmers who are familiar with either the Windows 95, Windows NT, or HP-UX environment. If you will be performing the VISA installation and configuration on Windows NT or HP-UX, you must also have either system administration privileges on your Windows NT system, or super-user (`root`) privileges on your HP-UX system.

Other Documentation

The following documentation is also helpful when using VISA:

- *HP I/O Libraries Installation and Configuration Guide* explains how to install and configure the HP VISA library and the HP Standard Instrument Control Library (SICL) on Microsoft Windows or HP-UX.
- *HP VISA Quick Reference Guide for C Programmers* helps you find VISA function syntax information quickly.
- *HP VISA Online Help* is provided in the form of Windows Help on Microsoft Windows, and in the form of manual pages (man pages) and online help on HP-UX.
- *HP VISA Example Programs* are provided online to help you develop your VISA applications more easily.

The following documents may also be helpful when using VISA:

- VXI*plug&play* System Alliance *VISA Library Specification 4.3*
- *IEEE Standard Codes, Formats, Protocols, and Common Commands* - ANSI/IEEE Standard 488.2-1992

The following VXIbus Consortium specifications may also be helpful when using VISA over LAN:

- *TCP/IP Instrument Protocol Specification* - VXI-11, Rev. 1.0
- *TCP/IP-VXIbus Interface Specification* - VXI-11.1, Rev. 1.0
- *TCP/IP-IEEE 488.1 Interface Specification* - VXI-11.2, Rev. 1.0
- *TCP/IP-IEEE 488.2 Instrument Interface Specification* - VXI-11.3, Rev. 1.0

Where to Go Next

Now that you have a better understanding of VISA, continue with one of the following chapters:

- Chapter 2, "Building an HP VISA Application in Windows"
- Chapter 3, "Building an HP VISA Application in HP-UX"

2

Building an HP VISA Application in Windows

Building an HP VISA Application in Windows

This chapter describes what you need to know to build a VISA application in a Windows environment. This chapter contains the following sections:

- Reviewing an HP VISA Program
- Linking to HP VISA Libraries
- Compiling and Linking an HP VISA Program
- Logging Error Messages
- Running an HP VISA Program
- Where to Go Next

Reviewing an HP VISA Program

In this section, you will first review a simple example program called `idn` that queries an HP-IB instrument for its identification string. This example uses the QuickWin or EasyWin feature of Microsoft and Borland C or C++ compilers on Windows.

The `idn` example files are located in the following subdirectories.

32-bit VISA on Windows 95:

```
\VXIPNP\WIN95\HPVISA\SAMPLES
```

32-bit VISA on Windows NT:

```
\VXIPNP\WINNT\HPVISA\SAMPLES
```

16-bit VISA on Windows 95:

```
\VXIPNP\WIN\HPVISA\SAMPLES
```

The source file `idn.c` is listed on the following page. An explanation of the various function calls in the example is provided directly after the program listing for your review.

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

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

void main () {

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

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

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

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

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

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

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

Example Program Contents

The following is a summary of the VISA function calls used in the example program. For a more detailed explanation of VISA functionality, see Chapter 4, "Programming with HP VISA."

`visa.h` This file is included at the beginning of the file to provide the function prototypes and constants defined by VISA.

`ViSession` The `ViSession` is a VISA data type. Each object that will establish a communication channel must be defined as `ViSession`.

`viOpenDefaultRM` You must first open a session with the default resource manager with the `viOpenDefaultRM` function. This function will initialize the default resource manager and return a pointer to that resource manager session.

`viOpen` This function establishes a communication channel with the device specified. A session identifier that can be used with other VISA functions is returned. This call must be made for each device you will be using.

`viPrintf` and `viScanf` These are the VISA formatted I/O functions that are patterned after those used in the C programming language. The `viPrintf` call sends the IEEE 488.2 `*RST` command to the instrument and puts it in a known state. The `viPrintf` call is used again to query for the device identification (`*IDN?`). The `viScanf` call is then used to read the results.

`viClose` This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.

Refer to Chapter 7, "HP VISA Language Reference," for more detailed information on these VISA function calls and to learn about all of the functions provided by VISA.

Linking to HP VISA Libraries

Your application must link to one of the VISA import libraries, as follows.

32-bit VISA on Windows 95:

`C:\VXIPNP\WIN95\LIB\MSC\VISA32.LIB` for Microsoft compilers
`C:\VXIPNP\WIN95\LIB\BC\VISA32.LIB` for Borland compilers

32-bit VISA on Windows NT:

`C:\VXIPNP\WINNT\LIB\MSC\VISA32.LIB` for Microsoft compilers
`C:\VXIPNP\WINNT\LIB\BC\VISA32.LIB` for Borland compilers

16-bit VISA on Windows 95:

`C:\VXIPNP\WIN\LIB\MSC\VISA.LIB` for Microsoft compilers
`C:\VXIPNP\WIN\LIB\BC\VISA.LIB` for Borland compilers

See the following section, "Compiling and Linking an HP VISA Program," for information on how to use the VISA run-time libraries.

32-bit Applications

The following is a summary of important compiler-specific considerations for several C/C++ compiler products when developing WIN32 applications.

For Microsoft Visual C++ version 2.0 compilers:

- Select **Project | Update All Dependencies** from the menu.
- Select **Project | Settings** from the menu. Click on the **C/C++** button. Select **Code Generation** from the **Category** list box and select **Multi-Threaded using DLL** from the **Use Run-Time Libraries** list box. VISA requires these definitions for WIN32. Click on **OK** to close the dialog boxes.
- Select **Project | Settings** from the menu. Click on the **Link** button and add **visa32.lib** to the **Object / Library Modules** list box. Optionally, you may add the library directly to your project file. Click on **OK** to close the dialog boxes.
- You may wish to add the include file and library file search paths. They are set by doing the following:
  1. Select **Tools | Options** from the menu.
  2. Click on the **Directories** button to set the include file path.
  3. Select **Include Files** from the **Show Directories For** list box.
  4. Click on the **Add** button and type in one of the following:

     C:\VXIPNP\WIN95\INCLUDE

     Or:

     C:\VXIPNP\WINNT\INCLUDE
  5. Select **Library Files** from the **Show Directories For** list box.

6. Click on the **Add** button and type in one of the following:

   ```
   C:\VXIPNP\WIN95\LIB\MSC
   ```

   Or:

   ```
   C:\VXIPNP\WINNT\LIB\MSC
   ```

For Borland C++ version 4.0 compilers:

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

  ```
  C:\VXIPNP\WIN95\INCLUDE
  C:\VXIPNP\WIN95\LIB\BC
  ```

  Or:

  ```
  C:\VXIPNP\WINNT\INCLUDE
  C:\VXIPNP\WINNT\LIB\BC
  ```

16-bit Applications

The following is a summary of important compiler-specific considerations for several C/C++ compiler products when developing WIN16 applications.

For Microsoft Visual C++ version 1.5 compilers:

- To set the memory model, do the following:
  1. Select **Options | Project**.
  2. Click on the **Compiler** button, then select **Memory Model** from the **Category** list.
  3. Click on the **Model** list arrow to display the model options, and select **Large**.
  4. Click on **OK** to close the **Compiler** dialog box.

- You may wish to add the include file and library file search paths. They are set under the `Options | Directories` menu selection:

  ```
  C:\VXIPNP\WIN\INCLUDE
  C:\VXIPNP\WIN\LIB\MSC
  ```

  Otherwise, the library and include files should be explicitly specified in the project file.

For Borland C (or Turbo C) compilers:

- Make sure large memory model is selected:

  1. Select `Options | Project`.
  2. Double-click on `16-bit Compiler` in the `Topics` list box.
  3. Click on `Memory Model`.
  4. Change `Mixed Model Override` to `Large`.
  5. Click on `OK` to close the dialog box.

  You can do this from the command line environment by specifying the `/ml` option to the compiler.

- The Borland C linker defaults to being case-insensitive when resolving references. To link to the VISA libraries, you will need to tell the linker to be case-sensitive for exports.

  To do this from Borland's Integrated Development Environment:

  1. Select `Options | Project`.
  2. Double-click on `Linker` in the `Topics` list box.
  3. Click on `General` in the `Topics` list box.
  4. Select `Case Sensitive exports and imports`.
  5. Click on `OK` to close the dialog box.

  You can do this from the command line environment by specifying the `/C` option to TLINK.

- You may wish to add the include file and library file search paths. They are set under the `Options | Project` menu selection. Double-click on `Directories` from the `Topics` list box and add:

```
C:\VXIPNP\WIN\INCLUDE
C:\VXIPNP\WIN\LIB\BC
```

- The following is required for building Borland EasyWin programs:

```
#if defined (_BORLANDC_) && !defined(_WIN32_)
  _InitEasyWin();
#endif
```

Logging Error Messages

Windows 95

While developing or debugging your VISA application, you may wish to view internal VISA messages while your application is running. This can be done by using the **Message Viewer** utility in the **HP I/O Libraries** program group on Windows 95. This utility provides a debug window to which VISA logs internal messages during application execution. Some of these internal messages do not represent programming errors and are actually error messages from VISA which are being handled internally by VISA.

To start the utility, double-click on the **Message Viewer** icon in the **HP I/O Libraries** program group. The utility must be started before execution of the VISA application. It will receive messages while minimized, however. The **Message Viewer** utility also provides menu selections for saving the logged messages to a file, and for clearing the message buffer.

Windows NT

VISA logs internal messages as Windows NT events. While developing your VISA application or tracking down problems, you may wish to view these messages. You can do so by starting the **Event Viewer** utility in the **Administrative Tools** group. Both system and application messages can be logged to the **Event Viewer** from VISA. VISA messages are identified either by **SICL LOG**, or by the driver name (for example, **hp341i32**).

Running an HP VISA Program

To run the **idn** example program, do the following:

- If you use the command line interface:

  Select **File | Run** from the Windows Program Manager menu.

- If you use the Windows interface:

  - For Borland, select **Run | Run**.
  - For Microsoft, select **Project | Execute** or **Run | Go**.

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

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

If the program does not run, refer to the message logger for a list of run-time errors.

Where to Go Next

Now that you understand some basics of programming with VISA, continue on to Chapter 4, "Programming with HP VISA." Chapter 4 provides detailed example programs. It also contains information on sessions, addressing, interrupt handling, locking, and so forth.

3

Building an HP VISA Application in HP-UX

Building an HP VISA Application in HP-UX

This chapter describes what you need to know to build a VISA application on HP-UX version 10.20 or later. This chapter contains the following sections:

- Reviewing an HP VISA Program
- Compiling and Linking an HP VISA Program
- Logging Error Messages
- Running an HP VISA Program
- Getting Online Help
- Where to Go Next

Reviewing an HP VISA Program

In this section, you will first review a simple example program called **idn** that queries an HP-IB instrument for its identification string. The **idn** example program is located in the following subdirectory:

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

The source file **idn.c** is listed on the following page. An explanation of the various function calls in the example is provided directly after the program listing for your review.

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

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

void main () {

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

  /* Open session to GPIB device at address 22 */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM, "GPIB0::24::INSTR", VI_NULL,VI_NULL, &vi);

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

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

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

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

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

The Example Program Contents

The following is a summary of the VISA function calls used in the example program. For a more detailed explanation of VISA functionality, see Chapter 4, "Programming with HP VISA."

`visa.h` This file is included at the beginning of the file to provide the function prototypes and constants defined by VISA.

`ViSession` The `ViSession` is a VISA data type. Each object that will establish a communication channel must be defined as `ViSession`.

`viOpenDefaultRM` You must first open a session with the default resource manager with the `viOpenDefaultRM` function. This function will initialize the default resource manager and return a pointer to that resource manager session.

`viOpen` This function establishes a communication channel with the device specified. A session identifier that can be used with other VISA functions is returned. This call must be made for each device you will be using.

`viPrintf` and `viScanf` These are the VISA formatted I/O functions that are patterned after those used in the C programming language. The `viPrintf` call sends the IEEE 488.2 `*RST` command to the instrument and puts it in a known state. The `viPrintf` call is used again to query for the device identification (`*IDN?`). The `viScanf` call is then used to read the results.

`viClose` This function must be used to close each session. When you close a device session, all data structures that had been allocated for the session will be deallocated. When you close the default manager session, all sessions opened using that default manager session will be closed.

Refer to Chapter 7, "HP VISA Language Reference," for more detailed information on these VISA functions and to learn about all of the functions provided by VISA.

Compiling and Linking an HP VISA Program

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

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

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

Logging Error Messages

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

```
ErrorLog=true
```

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

Running an HP VISA Program

Execute your VISA program by typing the program name at the command prompt. For example:

```
idn
```

When using an HP 54601A Four Channel Oscilloscope, you should get something similar to the following:

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

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

Getting Online Help

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

Using the HyperHelp Viewer

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

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

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

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

Using HP-UX Manual Pages

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

man *function*

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

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

Where to Go Next

Now that you understand some basics of programming with VISA, continue on to Chapter 4, "Programming with HP VISA." Chapter 4 provides detailed example programs. It also contains information on sessions, addressing, interrupt handling, locking, and so forth.

4

Programming with HP VISA

Programming with HP VISA

This chapter describes how to program with VISA. The basics of VISA are described, including formatted I/O, events and handlers, attributes, and locking. Example programs are also provided and can be found in the **SAMPLES** subdirectory on Windows environments, or in the **examples** subdirectory on HP-UX. See Appendix A, "HP VISA System Information," for the specific location of the example programs on your operating system.

This chapter contains the following sections:

- Including the HP VISA Declarations File
- Opening a Session
- Addressing a Session
- Closing a Session
- Searching for Resources
- Sending I/O Commands
- Using Attributes
- Using Events and Handlers
- Trapping Errors
- Using Locks

For specific details on the VISA functions, see Chapter 7, "HP VISA Language Reference."

Including the HP VISA Declarations File

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. Note that the VISA types are also listed in Appendix D, "HP VISA Type Definitions."

Opening a Session

A session is a channel of communication. Sessions must first be opened on the default resource manager, and then for each device you will be using. The following is a summary of sessions that can be opened:

- 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 **device session** is used to communicate with a device on an interface. A device session must be opened for each device you will be using. When you use a device 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. Typically a device is an instrument, but could be a computer, a plotter, or a printer.

NOTE

All devices that you will be using need to be connected and in working condition prior to the first VISA function call (`viOpenDefaultRM`). The system is configured only on the *first* `viOpenDefaultRM` per process. Therefore, if `viOpenDefaultRM` is 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 in working condition.

Device Sessions

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

Next, you open a session with a specific device with the `viOpen` function. This function uses the session returned from `viOpenDefaultRM` and returns its own session to identify the device 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 device address specified in the *rsrcName* parameter to open a device session. The *vi* parameter in `viOpen` returns a session identifier that can be used with other VISA functions.

Your program may have several sessions open at the same time by creating multiple session identifiers by calling the `viOpen` function multiple times.

The following summarizes the parameters in the previous function calls:

sesn	This is a session returned from the `viOpenDefaultRM` function that identifies the resource manager session.
rsrcName	This is a unique symbolic name of the device (device address).
accessMode	This parameter is not used for VISA 1.0. Use `VI_NULL`.
timeout	This parameter is not used for VISA 1.0. Use `VI_NULL`.
vi	This is a pointer to the session identifier for this particular device session. This pointer will be used to identify this device session when using other VISA functions.

The following is an example of opening sessions with a GPIB multimeter and a GPIB-VXI scanner:

```
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);
```

The previous example first opens a session with the default resource manager. The session returned from the resource manager and a device address is then used to open a session with the GPIB device at address 22. That session will now be identified as **dmm** when using other VISA functions. The session returned from the resource manager is then used again with another device address to open a session with the GPIB-VXI device at primary address 9 and VXI logical address 24. That session will now be identified as **scanner** when using other VISA functions. See the following section, "Addressing a Session," for information on addressing particular devices.

Addressing a Session

As seen in the previous section, the *rsrcName* parameter in the `viOpen` function is used to identify a specific device. This parameter consists of the VISA interface name and the device address. The interface name is determined when you run the VISA configuration utility. This name is usually the interface type followed by a number. The following table illustrates the format of the *rsrcName* for the different interface types:

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

The following describes the parameters used above:

board	This optional parameter is used if you have more than one interface of the same type. The default value for *board* is 0.
VXI logical address	This is the logical address of the VXI instrument.
primary address	This is the primary address of the GPIB device.
secondary address	This optional parameter is the secondary address of the GPIB device. If no secondary address is specified, none is assumed.

`INSTR` is an optional parameter that indicates that you are communicating with a resource that is of type `INSTR`, meaning instrument.

NOTE

If you want to be compatible with future releases of VISA, you must include the **INSTR** parameter in the syntax.

The following are examples of valid symbolic names:

`VXI0::24::INSTR`	Device at VXI logical address 24 that is of VISA type INSTR.
`VXI2::128`	Device at VXI logical address 128, in the third VXI system (VXI2).
`GPIB-VXI0::24`	A VXI device at logical address 24. This VXI device is connected via a GPIB-VXI command module.
`GPIB0::7::0`	A GPIB device at primary address 7 and secondary address 0 on the GPIB interface.
`ASRL1::INSTR`	A serial device located on port 1 that is of VISA type INSTR.

The following is an example of opening a device session with the GPIB device at primary address 23.

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

Closing a Session

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

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

Searching for Resources

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

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

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

Where the parameters are defined as follows:

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

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

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

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

NOTE

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

The following example searches the VXI interface for devices. 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, and so on.

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

Note that **VI_FIND_BUFLEN** is defined in the **visa.h** declarations file.

Sending I/O Commands

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

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

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

Formatted I/O

The VISA formatted I/O mechanism is similar to the C `stdio` mechanism. The VISA formatted I/O functions are buffered. They are as follows:

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

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

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

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

- The `viQueryf` functions format and send data to a device and then immediately receive and convert the response data. Hence, the `viQueryf` function is a combination of the `viPrintf` and `viScanf` functions. Similarly, the `viVQueryf` function is a combination of the `viVPrintf` and `viVScanf` functions.

  The `viQueryf` function sends and receives separate *arg* parameters, while the `viVQueryf` function sends and receives a list of parameters in *params*:

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

There are two non-buffered and non-formatted I/O functions that synchronously transfer data called `viRead` and `viWrite`, and there are two that asynchronously transfer data called `viReadAsync` and `viWriteAsync`. These are raw I/O functions and do not intermix with the formatted I/O functions. See "Non-Formatted I/O" later in this chapter.

See `viPrintf`, `viQueryf`, and `viScanf` in Chapter 7, "HP VISA Language Reference," for more information on how data is converted under the control of the format string.

Formatted I/O Conversion

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

%[modifiers]conversion character

Modifiers

Zero or more modifiers may be used to change the meaning of the conversion character. Modifiers are only used when sending or receiving formatted I/O.

For sending formatted I/O, the asterisk (*****) can be used to indicate that the number is taken from the next argument. However, when the asterisk is used when receiving formatted I/O, it indicates that the assignment is suppressed and the parameter is discarded. Use the pound sign (**#**) when receiving formatted I/O to indicate that an extra argument is used.

The following are supported modifiers.

Field Width. Field width is an optional integer that specifies how many characters are in the field. If the **viPrintf** or **viQueryf** (*writeFmt*) formatted data has fewer characters than specified in the field width, it will be padded on the left, or on the right if the – flag is present. You can use an asterisk (*****) in place of the integer in **viPrintf** or **viQueryf** (*writeFmt*) to indicate that the integer is taken from the next argument. For the **viScanf** or **viQueryf** (*readFmt*) functions, you can use a **#** sign to indicate that the next argument is a reference to the field width.

The field width modifier is only supported with **viPrintf** and **viQueryf** (*writeFmt*) conversion characters **d**, **f**, **s**, and **viScanf** and **viQueryf** (*readFmt*) conversion characters **c**, **s**, and **[]**.

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

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

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

. **Precision.** Precision is an optional integer preceded by a period. This modifier is only used with the `viPrintf` and `viQueryf` (*writeFmt*) functions. The meaning of this argument is dependent on the conversion character used:

Precision Modifiers

Conversion Character	Description
`d`	Indicates that the minimum number of digits to appear is specified for the `@1`, `@H`, `@Q`, and `@B` flags, and the `i`, `o`, `u`, `x`, and `X` conversion characters.
`f`	Indicates that the maximum number of digits after the decimal point is specified.
`s`	Indicates that the maximum number of characters for the string is specified.
`g`	Indicates that the maximum significant digits are specified.

You can use an asterisk (`*`) in place of the integer to indicate that the integer is taken from the next argument.

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

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

Sends : `26.93`

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

Argument Length Modifiers

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

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

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

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

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

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

See the **viPrintf** function in Chapter 7, "HP VISA Language Reference," for additional, enhanced modifiers you may use (**@1**, **@2**, **@3**, **@H**, **@Q**, and **@B**).

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

\n	Sends the ASCII line feed character. The END identifier will also be sent.
\r	Sends an ASCII carriage return character.
\t	Sends an ASCII TAB character.
\###	Sends the ASCII character specified by the octal value.
\"	Sends the ASCII double quote character.
****	Sends a backslash character.

NOTE

Note that the * while using the **viScanf** functions acts as an assignment suppression character. The input is not assigned to any parameters and is discarded.

Conversion Characters. The conversion characters for sending and receiving formatted I/O are different. The following tables summarize the conversion characters for each:

viPrintf/viVPrintf and viQueryf/viVQueryf (*writeFmt*)
Conversion Characters

Conversion Character	Description
d, i	Corresponding argument is an integer.
f	Corresponding argument is a double.
c	Corresponding argument is a character.
s	Corresponding argument is a pointer to a null terminated string.
%	Sends an ASCII percent (%) character.
o,u,x,X	Corresponding argument is an unsigned integer.
e,E,g,G	Corresponding argument is a double.
n	Corresponding argument is a pointer to an integer.
b, B	Corresponding argument is the location of a block of data.

viScanf/viVScanf and viQueryf/viVQueryf (*readFmt*)
Conversion Characters

Conversion Character	Description
d,i,n	Corresponding argument must be a pointer to an integer.
e,f,g	Corresponding argument must be a pointer to a float.
c	Corresponding argument is a pointer to a character sequence.
s,t,T	Corresponding argument is a pointer to a string.
o,u,x	Corresponding argument must be a pointer to an unsigned integer.
[	Corresponding argument must be a character pointer.
b	Corresponding argument is a pointer to a data array.

The following example receives data from the session specified by the *vi* parameter and converts the data to a string:

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

Formatted I/O Example

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

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

```
/*formatio.c
  This example program makes a multimeter measurement with a comma
  separated list passed with formatted I/O and prints the results.
  Note that you must change the device address. */

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

void main () {

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

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

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

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

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

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

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

Format String

The format string for `viPrintf` and `viQueryf` (*writeFmt*) puts a special meaning on the newline character (`\n`). The newline character in the format string flushes the output buffer to the device. All characters in the output buffer will be written to the device with an END indicator included with the last byte (the newline character). This means that you can control at what point you want the data written to the device. If no newline character is included in the format string, then the characters converted are stored in the output buffer. It will require another call to `viPrintf`, `viQueryf` (*writeFmt*), or `viFlush` to have those characters written to the device.

This can be very useful in queuing up data to send to a device. It can also raise I/O performance by doing a few large writes instead of several smaller writes.

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.

The write buffer is maintained by the `viPrintf` or `viQueryf` (*writeFmt*) functions. The buffer queues characters to send to the device so that they are sent in large blocks, thus increasing performance. The write buffer automatically flushes when it sends a newline character from the format string. It may occasionally be flushed at other non-deterministic times, such as when the buffer fills. When the write buffer flushes, it sends its contents to the device. If you set the `VI_ATTR_WR_BUF_OPER_MODE` attribute to `VI_FLUSH_ON_ACCESS`, the write buffer will also be flushed every time a `viPrintf` or `viQueryf` operation completes. See "Using Attributes" later in this chapter for information on setting VISA attributes.

The read buffer is maintained by the `viScanf` and `viQueryf` (*readFmt*) functions. It queues the data received from a device until it is needed by the format string. Flushing the read buffer destroys the data in the buffer and guarantees that the next call to `viScanf` or `viQueryf` reads data directly from the device rather than data that was previously queued. If you set the `VI_ATTR_RD_BUF_OPER_MODE` attribute to `VI_FLUSH_ON_ACCESS`, the read buffer will be flushed every time a `viScanf` or `viQueryf` operation completes. See "Using Attributes" later in this chapter for information on setting VISA attributes.

You can manually flush the read and write buffers by using the `viFlush` function.

NOTE

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.

You can modify the size of the buffer by using the **viSetBuf** function. See Chapter 7, "HP VISA Language Reference," for more information on this function.

Non-Formatted I/O

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

The non-formatted I/O functions are as follows:

- 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*);

- 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*);

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

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

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

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

For more information, see the **viRead, viWrite, viReadAsync, viWriteAsync,** and **viTerminate** functions in Chapter 7, "HP VISA Language Reference."

Non-Formatted I/O Example

The following example program illustrates using non-formatted I/O functions to communicate with a GPIB device. A similar example is used to illustrate formatted I/O earlier in this chapter. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" later in this chapter.

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

```
/*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("Measurement Results: %s\n", strres);

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

Using Attributes

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

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 following example reads the state of the **VI_ATTR_TERMCHAR_EN** attribute and changes it if it is not true:

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

NOTE

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

The attributes are described in the following subsections. For programming information on attributes, such as attribute types and ranges, see Appendix B, "HP VISA Attributes."

HP VISA Resource Attributes

The VISA resource attributes are primarily used to find out information about the VISA version implemented and its manufacturer. Information can also be obtained about the current resource manager session, as well as the locking state of a resource.

`VI_ATTR_MAX_QUEUE_LENGTH`	Specifies the maximum number of events that can be queued.
`VI_ATTR_RM_SESSION`	Returns the session of the resource manager that was used to open this session.
`VI_ATTR_RSRC_IMPL_VERSION`	Returns the resource identification.
`VI_ATR_RSRC_LOCK_STATE`	Returns the current locking state of the resource.
`VI_ATTR_RSRC_MANF_ID`	Returns the VXI manufacturer's identification of the manufacturer that created the implementation.
`VI_ATTR_RSRC_MANF_NAME`	Returns the VXI manufacturer's name of the manufacturer that created the implementation.
`VI_ATTR_RSRC_NAME`	Returns the identifier of the resource compliant with the address specified.
`VI_ATTR_RSRC_SPEC_VERSION`	Returns the VISA version.
`VI_ATTR_USER_DATA`	This is a place for you to store your own data.

HP VISA Generic Instrument Attributes

The following are generic attributes that can be called on sessions. These attributes determine such things as when a buffer is flushed, timeout values, and the type of interface the device is on.

`VI_ATTR_INTF_NUM`	Returns the board number of the specified interface.
`VI_ATTR_INTF_TYPE`	Returns the interface type for the specified session.
`VI_ATTR_IO_PROT`	For VXI, specifies if you use normal word serial or fast data channel (FDC) protocol. For GPIB, only normal data transfers are accepted.
`VI_ATTR_RD_BUF_OPER_MODE`	Determines when the read buffer is flushed.
`VI_ATTR_SEND_END_EN`	Specifies whether the END is asserted during the transfer of the last byte of the buffer.
`VI_ATTR_SUPPRESS_END_EN`	Specifies whether the END is suppressed.
`VI_ATTR_TERMCHAR`	Specifies if the termination character is to be used. When `VI_ATTR_TERMCHAR_EN` is enabled and the termination character is read, the read operation will terminate.
`VI_ATTR_TERMCHAR_EN`	Determines if the read operation will terminate when a termination character is received.
`VI_ATTR_TMO_VALUE`	Specifies a timeout value.
`VI_ATTR_TRIG_ID`	Specifies the current trigger line.
`VI_ATTR_WR_BUF_OPER_MODE`	Determines when the write buffer is flushed.

HP VISA Interface Specific Instrument Attributes

The interface specific attributes provide information about an interface or a device on an interface. The attributes are listed by interface type.

GPIB and GPIB-VXI Interfaces	`VI_ATTR_GPIB_PRIMARY_ADDR`	Returns the primary address of the GPIB device for the specified session.
	`VI_ATTR_GPIB_SECONDARY_ADDR`	Returns the secondary address of the GPIB device for the specified session.
VXI and GPIB-VXI Interfaces	`VI_ATTR_CMDR_LA`	Returns the logical address of the commander of the VXI device in the specified session.
	`VI_ATTR_DEST_INCREMENT`	Specifies how much the destination offset is to be incremented after every transfer in the **`viMoveOut`***XX* function. If set to 0, the **`viMoveOut`***XX* function will always write to the same element, essentially treating the destination as a FIFO register.
	`VI_ATTR_FDC_CHNL`	Determines which fast data channel (FDC) will be used to transfer the buffer.
	`VI_ATTR_FDC_GEN_SIGNAL_EN`	Setting this attribute to `VI_TRUE` lets the servant send a signal when control of the FDC channel is passed back to the commander. This action frees the commander from having to poll the FDC header while engaging in an FDC transfer.
	`VI_ATTR_FDC_MODE`	Determines which FDC mode to use (Normal or Stream mode).
	`VI_ATTR_FDC_USE_PAIR`	If set to `VI_TRUE`, a channel pair will be used for transferring data. Otherwise, only one channel will be used.

`VI_ATTR_IMMEDIATE_SERV`	Specifies whether or not the given device is an immediate servant of the controller running VISA.
`VI_ATTR_MAINFRAME_LA`	Returns the lowest logical address in the mainframe. **`VI_UNKNOWN_LA`** is returned if the logical address is not known.
`VI_ATTR_MANF_ID`	Returns the manufacturer's identification number of the VXI device in the specified session.
`VI_ATTR_MEM_BASE`	Returns the base address of the device in A24 or A32 VXI memory address space.
`VI_ATTR_MEM_SIZE`	Returns the size of memory requested by the device in A24 or A32 VXI address space.
`VI_ATTR_MEM_SPACE`	Returns the VXI address space used by the device (A16, A16/A24, or A16/A32).
`VI_ATTR_MODEL_CODE`	Returns the model code of the device in the specified session.
`VI_ATTR_SLOT`	Returns the physical slot location of the VXI device in the specified session.
`VI_ATTR_SRC_INCREMENT`	Specifies how much the source offset is to be incremented after every transfer in the **`viMoveIn`***XX* function. If set to 0, the **`viMoveIn`***XX* function will always read from the same element, essentially treating the source as a FIFO register.
`VI_ATTR_VXI_LA`	Returns the logical address of the VXI device in the specified session.
`VI_ATTR_WIN_ACCESS`	Returns the mode in which the current window can be accessed.
`VI_ATTR_WIN_BASE_ADDR`	Returns the base address of the interface bus to which this window is mapped.
`VI_ATTR_WIN_SIZE`	Returns the size of the region mapped to this window.

GPIB-VXI Interface	VI_ATTR_INTF_PARENT_NUM	Returns the board number of the GPIB interface to which the GPIB-VXI is attached.
ASRL Interface	VI_ATTR_ASRL_AVAIL_NUM	Returns the number of bytes available in the global receive buffer.
	VI_ATTR_ASRL_BAUD	Returns the baud rate of the interface.
	VI_ATTR_ASRL_DATA_BITS	Returns the number of data bits contained in each frame (from 5 to 8). The data bits for each frame are located in the low-order bits of every byte stored in memory.
	VI_ATTR_ASRL_END_IN	Indicates the method used to terminate read operations.
	VI_ATTR_ASRL_END_OUT	Indicates the method used to terminate write operations.
	VI_ATTR_ASRL_FLOW_CNTRL	Returns the kind of flow control that the transfer mechanism is using.
	VI_ATTR_ASRL_PARITY	Returns the parity used with every frame transmitted and received.
	VI_ATTR_ASRL_STOP_BITS	Returns the number of stop bits used to indicate the end of a frame.

HP VISA Event Attributes

The following attributes are read only attributes that can only be read on event contexts returned from event handlers or `viWaitOnEvent`.

`VI_ATTR_EVENT_TYPE`	Returns the type of event enabled.
`VI_ATTR_SIGP_STATUS_ID`	Returns the 16-bit status (ID) value. (Only for `VI_EVENT_VXI_SIGP` event type.)
`VI_ATTR_RECV_TRIG_ID`	Returns which trigger line was fired. (Only for `VI_EVENT_TRIG` event type.)
`VI_ATTR_STATUS`	Returns the return code of the asynchronous I/O operation that has completed. (Only for `VI_EVENT_IO_COMPLETION` event type.)
`VI_ATTR_JOB_ID`	Returns the job identifier (ID) of the asynchronous operation that has completed. (Only for `VI_EVENT_IO_COMPLETION` event type.)
`VI_ATTR_BUFFER`	Returns the address of a buffer that was used in an asynchronous operation. (Only for `VI_EVENT_IO_COMPLETION` event type.)
`VI_ATTR_RET_COUNT`	Returns the actual number of elements that were asynchronously transferred. (Only for `VI_EVENT_IO_COMPLETION` event type.)

Using Events and Handlers

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.

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

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

These methods are independent of each other, and one or both can be used at one time. The callback method is generally used when immediate response is needed, and the queuing method is for non-critical events.

Examples of each of these methods follows. For a more detailed explanation of each method, see the following sections.

Callback Method:

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

   /* your event handling code here */

  viClose(context);
}
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);
.
}
```

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);
}
```

Events and Attributes

The following events can be enabled:

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

NOTE

The `VI_EVENT_VXI_SIGP` and `VI_EVENT_TRIG` events are *not* supported on the GPIB-VXI interface.

NOTE

Event contexts should *not* be closed in event handlers. (That is, do *not* use `viClose` to close contexts in event handlers.)

Once the application has received an event, information about that event can be obtained by using the `viGetAttribute` function on that particular event context. The following table lists the events and the associated read only attributes that can be read to get event information on a specific event.

Event Name	Attributes	Data Type	Values
VI_EVENT_SERVICE_REQ	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_REQ
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE VI_ATTR_SIGP_STATUS_ID	ViEventType ViUInt16	VI_EVENT_VXI_SIGP 0 to FFFFh
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	ViEventType ViStatus ViJobId ViBuf ViUInt32	VI_EVENT_IO_COMPLETION N/A N/A N/A 0 to FFFFFFFFh

Use the VISA `viReadSTB` function to read the status byte of the service request.

Reading the Attribute

Once you have decided which attribute to check, you can read the attribute using the `viGetAttribute` function. The following shows how you would check to find out which trigger line fired when the `VI_EVENT_TRIG` event was delivered:

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

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 "Using Attributes" earlier in this chapter for more information on reading attribute states.

The Callback Method

The callback method of event notification is used when you need to immediately respond to an event. To use the callback method for receiving notification that an event has occurred, you must do the following:

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

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

Installing Handlers

A handler is installed on a specified session. Only one handler can be installed on a specific event in a given session, or you can install a different handler for each event type. However, the same handler can be installed on more than one event type. Use the following function when installing an event handler:

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

Where the parameters are defined as follows:

vi	The session the handler will be installed on.
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*.

The following 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 in order for the service request event to be delivered.

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

Use the **viUninstallHandler** function to uninstall a specific handler. Or you can use wildcards (**VI_ANY_HNDLR** in the *handler* parameter) to uninstall groups of handlers. See **viUninstallHandler** in Chapter 7, "HP VISA Language Reference," for more details on this function.

Writing the Handler

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

Enabling Events

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

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

Where the parameters are defined as follows:

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. It can be enabled in several different ways:

- 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.

NOTE

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

context — Not used in VISA 1.0. Use `VI_NULL`.

The following 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.

Event Callback Example

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

This example program is installed on your system in the `SAMPLES` subdirectory on Windows environments, or in the `examples` subdirectory on HP-UX. See Appendix A, "HP VISA System Information," for the specific location of the example programs on your operating system.

```
/* evnthdlr.c
   This example program illustrates installing an event handler to
   be called when a trigger interrupt occurs. Note that you 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);
}
```

SRQ Callback Example

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

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

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

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

int srqOccurred;

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

   ViUInt16 statusByte;

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

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

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

    srqOccurred = 1;
    return VI_SUCCESS;
}

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

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

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

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

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

    srqOccurred = 0;

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

    /* Wait a while for the SRQ to be generated and for the handler */
    /* to be called. Print something while we wait */
    printf( "Waiting for an SRQ to be generated ." );
    for ( count = 0 ; (count < 10) && (srqOccurred == 0) ; count++ ) {
        long count2 = 0;
        printf( "." );
        while ( (count2++ < 100) && (srqOccurred ==0) ){
            YIELD;
        }
    }
    printf( "\n" );
```

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

    /* Clean up 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" );
}
```

The Queuing Method

The queuing method is generally used when you do not need immediate response from your application. 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, then the event information is retrieved and the program returns immediately. If the specified event has not occurred, then the program suspends execution until a specified event occurs or until the specified timeout period is reached.

Enabling Events

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

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

Where the parameters are defined as follows:

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.

The following 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*`);`

Note that the event must have previously been enabled with `VI_QUEUE` specified as the *mechanism* parameter.

The following 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.

Event Queuing Example

The following example 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 example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" later in this chapter.

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

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

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


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

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

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

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

   /* fire trigger line, twice */
   viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
   viAssertTrigger(vi, VI_TRIG_PROT_SYNC);
   /* Wait for the event to occur */
   err=viWaitOnEvent(vi, VI_EVENT_TRIG, 10000, &eventType, &eventVi);
```

```
    if(err==VI_ERROR_TMO){
       printf("Timeout Occurred! Event not received.\n");
       return;
    }

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

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

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

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

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


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

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

Trapping Errors

HP VISA 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 applications. To trap VISA errors you must check for `VI_SUCCESS` after each VISA function call. The following illustrates checking for `VI_SUCCESS`. If `VI_SUCCESS` is not returned, then an error handler, written by the programmer, is called. This must be done with each VISA function call.

NOTE

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);
```

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

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;
}
```

Instrument Errors

When programming instruments, it's good practice to check the instrument to make sure there are no instrument errors after each instrument function. The following function 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);
}
```

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

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

If a VISA resource is not locked by any of its sessions, all sessions have full privilege to invoke any operation and update any global attributes. Sessions are *not* required to have locks to invoke operations or update global attributes. However, if some other session has already locked the resource, attempts to update global attributes or invoke certain operations will fail. Refer to descriptions of the individual VISA functions in Chapter 7, "HP VISA Language Reference," to determine which would fail when a resource is locked.

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. The following subsection explains the different types, or access modes, of locks.

NOTE

The `viLock` and `viUnlock` functions are *not* supported with 16-bit VISA on Windows 95.

NOTE

If a resource is locked and the current session does not have the lock, the error `VI_ERROR_RSRC_LOCKED` is returned.

Lock Types

VISA defines two different types of locks:

- **Exclusive Lock** - A session can lock a VISA resource using the lock type `VI_EXCLUSIVE_LOCK` to get exclusive access privileges to the resource. This exclusive lock type excludes access to the resource from all other sessions. If a session has an exclusive lock, other sessions cannot modify global attributes or invoke operations on the resource; however, the other sessions *can* still get atttributes.
- **Shared Lock** - A session can share a lock on a VISA resource with other sessions by using the lock type `VI_SHARED_LOCK`. Shared locks in VISA are similar to exclusive locks in terms of access privileges, but can still be shared between multiple sessions. If a session has a shared lock, other sessions that share the lock can also modify global attributes and invoke operations on the resource (of course, unless some other session has a previous exclusive lock on that resource). A session that does not share the lock will lack these capabilities.

  See the next subsection, "Lock Sharing," for more information about the shared lock type.

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, note that VISA allows one of the sessions to acquire an exclusive lock along with the shared lock it is holding. This is explained in detail later in this section. Also note that VISA supports nested locking — that is, a session can lock the same VISA resource multiple times (for the same lock type) via multiple invocations of the `viLock` function. In such a case, unlocking the resource requires an equal number of invocations of the `viUnlock` function. Nested locking is also explained in detail later in this section.

NOTE

Some VISA operations may be permitted even when there is an exclusive lock on a resource, or some global attributes may not be read when there is any kind of lock on the resource. These exceptions, when applicable, are mentioned in the descriptions of the individual VISA functions and attributes. See Chapter 7, "HP VISA Language Reference," for descriptions of the individual functions to determine which are applicable for locking and which are not restricted by locking.

Lock Sharing

Because the locking mechanism in VISA is session-based, multiple threads sharing a session that has locked a VISA resource have the same privileges for accessing the resource. Some applications, though, may have separate sessions to a resource and may want all the sessions in that application to have the same privilege as the session that locked the resource. In other cases, there may be a need to share locks among sessions in different applications. Essentially, a session that acquired a lock to a resource may share the lock with other sessions it selects, and exclude access from other sessions.

As explained earlier, VISA defines the `VI_SHARED_LOCK` lock type to give exclusive access privileges to a session along with the capability to share these exclusive privileges with other sessions at the discretion of the original session. When locking the resource using the `VI_SHARED_LOCK` lock type, the `viLock` function returns an *accessKey* that can be used to share the lock. The session can then share this lock with any other session by passing around this *accessKey*.

Before other sessions can access the locked resource, they need to acquire the lock by passing the *accessKey* in the *requestedKey* parameter of the `viLock` function. Invoking `viLock` with the same key will register the new session to have the same access privileges as the original session. The new session that acquired the access privileges through the sharing mechanism can also pass the *accessKey* to other sessions for sharing of the resource, and so forth. Of course, all the sessions sharing a resource via the shared lock should synchronize their accesses to maintain a consistent state of the resource.

VISA also provides the flexibility for the application(s) to specify a key to use as the *accessKey*, instead of VISA generating the *accessKey*. The application(s) can suggest a key value to use through the *requestedKey* parameter of the `viLock` function. If the resource was not locked, the resource will use this *requestedKey* as the *accessKey*. If the resource was locked using a shared lock, and the *requestedKey* matches the key with which the resource was locked, the resource will grant shared access to the session. If an application attempts to lock a resource using a shared lock, but passes `VI_NULL` as the *requestedKey* parameter, then VISA will generate an *accessKey* for the session.

A session seeking to share exclusive access to a resource with other sessions needs to acquire a `VI_SHARED_LOCK` for this purpose. If it requests `VI_EXCLUSIVE_LOCK` instead, no valid *accessKey* will be returned. Consequently, the session will not be able to share the lock with any other sessions.

Acquiring an Exclusive Lock While Holding a Shared Lock

When multiple sessions have acquired a shared lock on a resource, VISA allows one of the sessions to acquire an exclusive lock along with the shared lock it is holding via the `viLock` function. The session holding both the exclusive and shared lock will have the same access privileges that it had when it was holding only the shared lock. However, this precludes the other sessions holding the shared lock from accessing the locked resource. This is useful when multiple sessions holding a shared lock must synchronize operations, or when one of the sessions must execute a critical operation.

When the session holding the exclusive lock unlocks the resource via the `viUnlock` function, all the sessions (including the one that had acquired the exclusive lock) will again have all the access privileges associated with the shared lock.

Note that in the reverse case where a session is holding an exclusive lock only (no shared locks), VISA does *not* allow it to change to `VI_SHARED_LOCK`.

Nested Locks

VISA also supports nested locking, in which a session can lock the same VISA resource multiple times (for the same lock type) via multiple invocations of the `viLock` function. Unlocking the resource requires an equal number of invocations of the `viUnlock` operation. In other words, for each invocation of `viLock`, a lock count will be incremented, and for each invocation of `viUnlock`, the lock count will be decremented. A resource will be truly unlocked only when the lock count is 0 (zero).

Each session maintains a separate lock count for each type of lock. Therefore, repeated invocations of the `viLock` function for the same session will increase the appropriate lock count, depending on the type of lock requested. In the case of a shared lock, nesting `viLock` functions will return with the same *accessKey* every time. In the case of an exclusive lock, `viLock` will not return any *accessKey*, regardless of whether it is nested or not.

For nesting shared locks, VISA does not require an *accessKey* be passed in to invoke the `viLock` function. That is, a session does not need to pass in the *accessKey* obtained from the previous invocation of `viLock` to gain a nested lock on the resource. However, if an application *does* pass in an *accessKey* when nesting shared locks, it must be the correct one for that session. See the description of the `viLock` function in Chapter 7, "HP VISA Language Reference," for further details on the *accessKey* parameter.

Lock Examples

The following two examples illustrate the two different lock types, exclusive and shared locks, in VISA. The first example shows a session gaining an exclusive lock to perform the `viPrintf` and `viScanf` VISA operations on a GPIB device. It then releases the lock via the `viUnlock` function.

```
/* lockexcl.c
  This example program queries a GPIB device for an identification string
  and prints the results. Note that you 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);
}
```

This second locking example shows a session gaining a shared lock with the *accessKey* called **lockkey**. Other sessions can now use this *accessKey* in the *requestedKey* parameter of the **viLock** function to share access on the locked resource. This example then shows the original session acquiring an exclusive lock while maintaining its shared lock. When the session holding the exclusive lock unlocks the resource via the **viUnlock** function, all the sessions sharing the lock again have all the access privileges associated with the shared lock.

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

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

void main () {

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

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

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

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

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

  /* Make sure no other process or thread does anything to this resource
     between the viPrintf() and the viScanf() calls
     NOTE:  this also locks out the processes with which we shared our
     'shared lock' key.
  */
  viLock (vi, VI_EXCLUSIVE_LOCK, 2000, VI_NULL, VI_NULL);
```

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

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

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

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

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

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

5

Programming VXI Devices

Programming VXI Devices

VISA supports three interfaces you can use to access VXI: 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. This chapter describes additional information for programming VXI devices with the VXI or GPIB-VXI interfaces. See Chapter 4, "Programming with HP VISA," for general information on VISA programming for all three interfaces.

This chapter contains the following sections:

- Programming Overview
- Using High-Level Memory Functions
- Using Low-Level Memory Functions
- Considering VXI Backplane Memory I/O Performance
- Using VXI Specific Attributes

For information on the specific VISA functions, see Chapter 7, "HP VISA Language Reference."

Programming Overview

You can use VISA to program VXI instruments over three different interfaces:

VXI Interface	Uses an embedded VXI controller or other VXI interfaces and accesses VXI instruments directly over the VXI backplane.
GPIB-VXI Interface	Uses the GPIB interface connected to a Command Module to directly access the VXI backplane.
GPIB Interface	Uses the GPIB interface connected to a Command Module and communicates with the Command Module, which then sends commands to the VXI instruments. There is no direct access to the VXI backplane.

This chapter discusses using the VXI and GPIB-VXI interfaces for direct access to the VXI backplane. When directly accessing the VXI backplane, you must be aware of the different types of VXI instruments:

Message-Based

A message-based device has its own processor which allows it to interpret the high-level commands, such as SCPI (Standard Commands for Programmable Instruments). While using VISA, you can simply place the SCPI command within your VISA output function call, and then the message-based device interprets the SCPI command. In this case you can use the VISA formatted I/O or non-formatted I/O functions and program the message-based device as you would a GPIB device. However, if your 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 that you can use to program directly to the registers: high-level memory functions or low-level memory functions. Each of these programming methods is discussed in the following sections.

Register-Based

A register-based device typically does not have a processor to interpret high-level commands; therefore, it must be programmed with register peeks and pokes directly to the device's registers. VISA provides two different methods that you can use to program register-based devices: high-level memory functions or low-level memory functions. Each of these programming methods is discussed in the following sections.

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, use the high-level memory functions, and the memory mapping and direct register access is done for you.

The trade off, however, is speed. The high-level memory functions are easier to use. Yet, because these functions encompass mapping of memory space and direct register access, the associated overhead slows down the program's execution time. If speed is what you need, use the low-level memory functions discussed in the next section.

The high-level memory functions include the `viIn` and `viOut` functions for transferring 8-, 16-, or 32-bit values, as well as the `viMoveIn` and `viMoveOut` functions for transferring 8-, 16-, or 32-bit blocks of data to or from local memory.

Programming to the Registers

When using the `viIn` and `viOut` high-level memory functions to program to the device registers, all you have to do is specify the session identifier, address space, and the offset of the register. The 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. The following are valid *space* values:

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

The *val32* parameter is a pointer to where the data read will be stored. If, instead, you were writing to the registers via the `viOut32` function, the *val32* parameter would be a pointer to the data to write to the specified registers.

NOTE

If the device specified by *vi* does not have memory in the specified address space, an error is returned.

The following is an example of using `viIn16`:

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

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** functions moves an 8-, 16-, or 32-bit block of data from the specified offset to local memory, whereas the **viMoveOut** functions moves an 8-, 16-, or 32-bit block of data from local memory to the specified offset. Again, the memory mapping is done for you. For example, in this function:

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

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

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

You can therefore program using 8-, 16-, or 32-bit transfers. The following table summarizes the high-level memory functions.

Function	Description
`viIn8(`*vi, space, offset, val8*`);`	Reads 8 bits of data from the specified offset.
`viIn16(`*vi, space, offset, val16*`);`	Reads 16 bits of data from the specified offset.
`viIn32(`*vi, space, offset, val32*`);`	Reads 32 bits of data from the specified offset.
`viOut8(`*vi, space, offset, val8*`);`	Writes 8 bits of data to the specified offset.
`viOut16(`*vi, space, offset, val16*`);`	Writes 16 bits of data to the specified offset.
`viOut32(`*vi, space, offset, val32*`);`	Writes 32 bits of data to the specified offset.
`viMoveIn8(`*vi, space, offset, length, buf8*`);`	Moves an 8-bit block of data from the specified offset to local memory.
`viMoveIn16(`*vi, space, offset, length, buf16*`);`	Moves a 16-bit block of data from the specified offset to local memory.
`viMoveIn32(`*vi, space, offset, length, buf32*`);`	Moves a 32-bit block of data from the specified offset to local memory.
`viMoveOut8(`*vi, space, offset, length, buf8*`);`	Moves an 8-bit block of data from local memory to the specified offset.
`viMoveOut16(`*vi, space, offset, length, buf16*`);`	Moves a 16-bit block of data from local memory to the specified offset.
`viMoveOut32(`*vi, space, offset, length, buf32*`);`	Moves a 32-bit block of data from local memory to the specified offset.

High-Level Memory Functions Examples

The following example programs use the high-level memory functions to read the ID and Device Type registers of the device at the VXI logical address of 24. The contents of the registers are then printed out. The first program uses the VXI interface, and the second program accesses the backplane with the GPIB-VXI interface. Note that these two programs are identical except for the string passed to `viOpen`.

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

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

void main () {

  ViSession defaultRM, dmm;
  unsigned short id_reg, devtype_reg;

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

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

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

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

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

The following example program uses the GPIB-VXI interface for direct register access through a VXI Command Module.

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

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

void main () {

  ViSession defaultRM, dmm;
  unsigned short id_reg, devtype_reg;

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

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

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

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

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

Using Low-Level Memory Functions

Low-level memory functions allow you direct access to memory on the interface just as with 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 to 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. Then you can send a series of peeks and pokes using the **viPeek** and **viPoke** low-level memory functions. When you are done, you must free the address window using the **viUnmapAddress** function. In sum, the process you might follow is:

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

Mapping Memory Space

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

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

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

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

A pointer to the address space where the memory was mapped is returned in the *address* parameter.

NOTE

If the device specified by *vi* does not have memory in the specified address space, an error is returned.

The following are example **viMapAddress** function calls:

```
/* 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 the Device Registers

Once 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's 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.

The following is an example of using `viPeek16`:

```
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)
```

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

Function	Description
viMapAddress(*vi, mapSpace, mapBase, mapSize, access, suggested, address*);	Maps the specified memory space.
viPeek8(*vi, addr, val8*);	Reads 8 bits of data from the address specified.
viPeek16(*vi, addr, val16*);	Reads 16 bits of data from the address specified.
viPeek32(*vi, addr, val32*);	Reads 32 bits of data from the address specified.
viPoke8(*vi, addr, val8*);	Writes 8 bits of data to the address specified.
viPoke16(*vi, addr, val16*);	Writes 16 bits of data to the address specified.
viPoke32(*vi, addr, val32*);	Writes 32 bits of data to the address specified.
viUnmapAddress(*vi*);	Unmaps memory space previously mapped.

Unmapping Memory Space

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

Low-Level Memory Functions Examples

The following example programs 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. Note that these two programs are identical except for the string passed to `viOpen`.

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

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

void main () {

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

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

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

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

  /* Read device type register contents */
  /* ViAddr is defined as a void * so we must cast it to something else */
  /* 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);
}
```

This example program uses the GPIB-VXI interface for direct register access through a VXI Command Module.

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

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

void main () {

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

  /* Open session to VXI device at address 24 */
  viOpenDefaultRM(&defaultRM);
  viOpen(defaultRM, "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);
}
```

Considering VXI Backplane Memory I/O Performance

VISA supports three different memory I/O mechanisms for accessing memory on the VXI backplane:

- Low-level **viPeek/viPoke**:
  - **viMapAddress**
  - **viUnmapAddress**
  - **viPeek8, viPeek16, viPeek32**
  - **viPoke8, viPoke16, viPoke32**
- High-level **viIn/viOut**:
  - **viIn8, viIn16, viIn32**
  - **viOut8, viOut16, viOut32**
- High-level **viMoveIn/viMoveOut**:
  - **viMoveIn8, viMoveIn16, viMoveIn32**
  - **viMoveOut8, viMoveOut16, viMoveOut32**

All three of these access mechanisms can be used to read and write VXI memory in the A16, A24, and A32 address spaces. The best method to use depends on the VISA program characteristics.

Low-level **viPeek/viPoke** is the most efficient in programs which require repeated access to different addresses in the same memory space. The advantages are:

- Individual **viPeek/viPoke** calls are faster than **viIn/viOut** or **viMoveIn/viMoveOut** calls.
- Memory pointer may be directly dereferenced in some cases for the lowest possible overhead. (See the example later in this section.)

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

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

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

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

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

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 by HP on each platform to provide the fastest possible transfer rate for large blocks of data. Note that for small blocks, the overhead associated with **viMoveIn/voMoveOut** may actually make these calls longer than an equivalent loop of **viIn/viOut** calls. The block size at which **viMoveIn/viMoveOut** becomes faster depends on the particular platform and processor speed. The advantages are:

- Simple to use.
- No limit on number of active maps.
- A16, A24, and A32 memory access can be mixed in a single *vi* session.
- Provides the best performance when transferring large blocks of data.
- Supports both block and FIFO mode.

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

The following is an example of 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 <stdio.h>

#define VXI_INST "VXI024INSTR"

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);

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

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

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

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

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

    /* Use direct memory dereferencing if 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);

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

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

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

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

/* ================================================================
   ================== High Level block memory I/O ==================
      = viMoveIn16
   The viMoveIn/viMoveOut commands do both block read/write and FIFO
   read write.

   These commands offer the best performance for reading and writing
   large data blocks on the VXI backplane.  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.

   ================================================================ */

   /*
      ================ Demonstrate block read ======================
      Read the instrument id register and device type register into
      an array.
   */
  viMoveIn16 (vi, VI_A16_SPACE, 0x00, 2, memArray);

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

```
    /*
       ================== Demonstrate FIFO read ========================
       First set the source increment to 0 so we will 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 VXI Specific Attributes

The VXI specific attributes can be useful to determine the state of your VXI system. There are read only and read/write attributes. The read only attributes specify things such as the logical address of the VXI device, and information about where your VXI device is mapped.

The following subsections show how you might use some of the VXI specific attributes. See Appendix B, "HP VISA Attributes," for programming information on the VISA attributes.

Using the Map Address as a Pointer

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

`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.

The following example shows how you can read the VI_ATTR_WIN_ACCESS attribute and use the result to determine how to access memory:

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

Setting the VXI Trigger Line

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

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

The above function sets the VXI trigger line to TTL trigger line 0 (`VI_TRIG_TTL0`). The following are valid VXI trigger lines:

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

Once you set a VXI trigger line, you can set up an event handler to be called when the trigger line fires. See "Using Events and Handlers" in Chapter 4 for more information on setting up an event handler.

NOTE

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 the viAssertTrigger function to assert software or hardware triggers. If VI_ATTR_TRIG_ID is VI_TRIG_SW, then 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.

6

Programming over LAN

Programming over LAN

This chapter describes how to use VISA over the LAN (Local Area Network). LAN is a natural way to extend the control of instrumentation beyond the limits of typical instrument interfaces. In order to communicate over the LAN, you must have configured the **`VISA LAN Client`** during the HP I/O Libraries configuration. See the *HP I/O Libraries Installation and Configuration Guide* for instructions.

NOTE

LAN is *not* supported with 16-bit VISA on Windows 95.

This chapter contains the following sections:

- Overview of the LAN
- Considering LAN Configuration and Performance
- Communicating with Devices over LAN
- Using Timeouts with LAN
- Using Signal Handling with LAN
- HP VISA Function Support with LAN

NOTE

To start the LAN server on a Windows 95 or Windows NT system, see the "Starting the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.

To stop the LAN server on a Windows 95 or Windows NT system, see the "Stopping the LAN Server" section of Chapter 2, "Installing and Configuring the HP I/O Libraries," in the *HP I/O Libraries Installation and Configuration Guide for Windows*.

Overview of the LAN

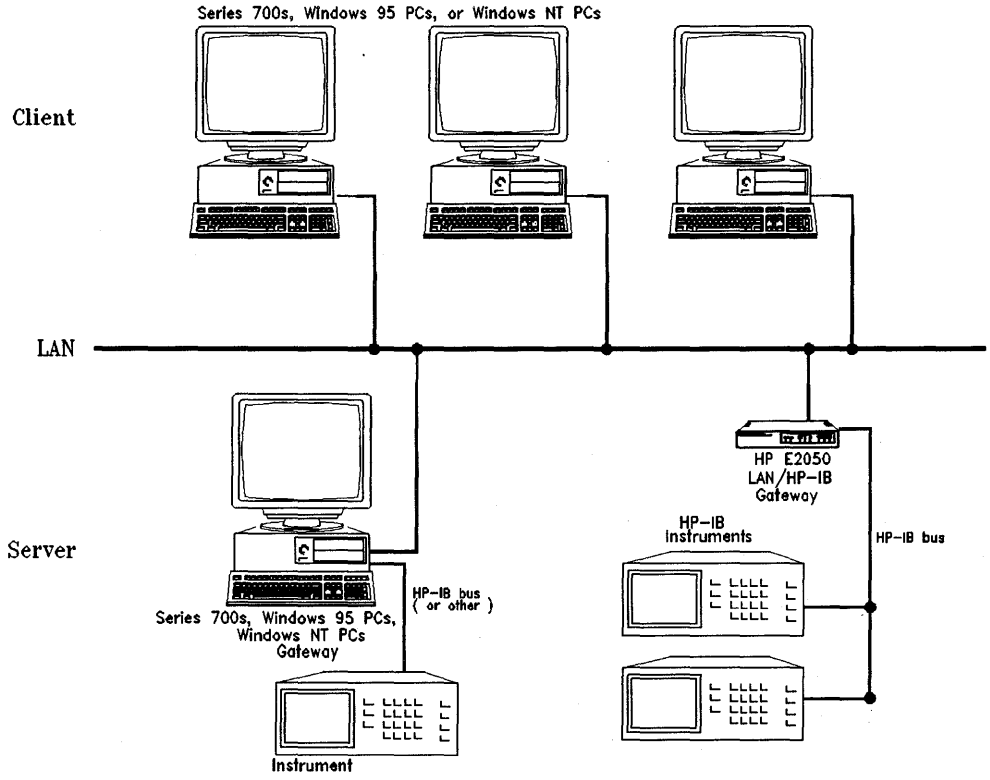
The LAN software provided with VISA allows you to control instrumentation over a LAN. LAN connections are included on many systems being sold today. By making use of these standard LAN connections, instrument control can be driven from a computer which does not have a special interface for instrument control.

The LAN software provided with VISA uses the client/server model of computing. **Client/server computing** refers to a model where an application, the **client**, does not perform all the necessary tasks of the application itself. Instead, the client makes requests of another computing device, the **server**, for certain services. Examples that you may have in your workplace include shared file servers, print servers, or database servers.

The use of LAN for instrument control also provides other advantages associated with client/server computing:

- Resource sharing by multiple applications/people within an organization.
- Distributed control, where the computer running the application controlling the devices need not be in the same room or even the same building as the devices themselves.

As shown in the following figure, a LAN client computer system (a Series 700 HP-UX workstation, a Windows 95 PC, or a Windows NT PC) makes VISA requests over the network to a LAN server (a Series 700 HP-UX workstation, a Windows 95 PC, a Windows NT PC, or an HP E2050 LAN/HP-IB Gateway). The LAN server is connected to the instrumentation or devices that must be controlled. Once the LAN server has completed the requested operation on the instrument or device, the LAN server sends a reply to the LAN client. This reply contains any requested data and status information which indicates whether the operation was successful.

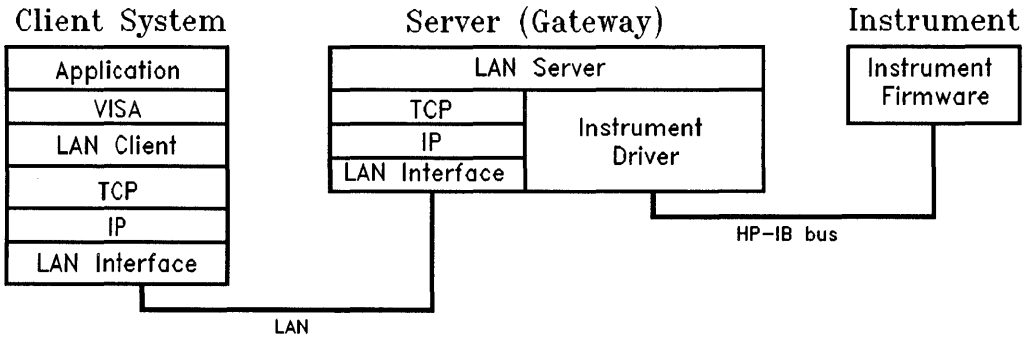


Using the LAN Client and LAN Server (Gateway)

The LAN server acts as a **gateway** between the LAN that your client system supports, and the instrument-specific interface that your device supports. Due to the LAN server's gateway functionality, we refer to devices or interfaces which are accessed via one of these LAN-to-instrument_interface gateways as being a LAN-gatewayed device or a LAN-gatewayed interface.

LAN Software Architecture

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



LAN Software Architecture

LAN Networking Protocols

The LAN software provided with VISA is built on top of standard LAN networking protocols. There are two LAN networking protocols provided with the VISA software. You can choose one or both of these protocols when configuring your systems (via the HP I/O Libraries configuration) to use VISA over LAN. The two protocols are as follows:

- **SICL LAN Protocol** is a networking protocol developed by HP which is compatible with all existing VISA LAN products. This LAN networking protocol is the default choice in the HP I/O Libraries configuration when you are configuring the LAN client. The SICL LAN protocol on HP-UX 10.20, Windows 95, and Windows NT currently supports VISA operations over the LAN to GPIB interfaces.
- **TCP/IP Instrument Protocol** is a networking protocol developed by the VXIbus Consortium based on the SICL LAN Protocol which permits interoperability of LAN software from different vendors that meet the VXIbus Consortium standards. Note that this LAN networking protocol may not be implemented with all the LAN products at this time. The TCP/IP Instrument Protocol on Windows 95 and Windows NT currently supports VISA operations over the LAN to GPIB interfaces.

When using either of these networking protocols, the LAN software provided with VISA uses the TCP/IP protocol suite to pass messages between the LAN client and the LAN server. The server accepts device I/O requests over the network from the client and then proceeds to execute those I/O requests on a local interface, such as HP-IB.

You can use both LAN networking protocols (SICL LAN Protocol and TCP/IP Instrument Protocol) with a LAN client. To do so, configure a LAN client and a VISA LAN client interface for each protocol, one specifying the SICL LAN Protocol and one specifying the TCP/IP Instrument Protocol. The LAN client and VISA LAN client are configured during the HP I/O Libraries configuration. (See the *HP I/O Libraries Installation and Configuration Guide* for information.)

Once you have configured VISA LAN client interfaces, one specifying SICL LAN Protocol, and one specifying TCP/IP Instrument Protocol, then you can use the interface name specified during configuration in your VISA `viOpen` call of your program. Note, however, that the LAN server does *not* support simultaneous connections from LAN clients using the SICL LAN Protocol and from other LAN clients using the TCP/IP Instrument Protocol.

LAN Client and Threads

You can use multi-threaded designs (where VISA calls are made from multiple threads) in WIN32 VISA applications over LAN. However, only one thread is permitted to access the LAN driver at a time. This sequential handling of individual threads by the LAN driver prevents multiple threads from colliding or overwriting one another. Note that requests are handled sequentially even if they are intended for different LAN servers.

If you want concurrent threads to be processed simultaneously with VISA over LAN, use multiple processes.

LAN Server

Currently there are three LAN servers that can be used with VISA: the HP E2050 LAN/HP-IB Gateway, an HP Series 700 running HP-UX, or a PC running Windows 95 or Windows NT. To use this capability, the LAN server must have a local HP-IB or GPIB interface configured for I/O. See the *HP I/O Libraries Installation and Configuration Guide* for information on configuration.

Note that the timing of operations performed remotely over a network will be different from the timing of operations performed locally. The extent of the timing difference will, in part, depend on the bandwidth of and the traffic on the network being used.

Contact your local HP representative for a current list of other HP supported LAN servers.

Considering LAN Configuration and Performance

As with other client/server applications on a LAN, when deploying an application which uses VISA over LAN, consideration must be given to the performance and configuration of the network to which the client and server will be attached. If the network to be used is not a dedicated LAN or otherwise isolated via a bridge or other network device, current utilization of the LAN must be considered. Depending on the amount of data which will be transferred over the LAN via the VISA application, performance problems could be experienced by the VISA application or other network users if sufficient bandwidth is not available. This is not unique to VISA over LAN, but is simply a general design consideration when deploying any client/server application.

If you have questions concerning the ability of your network to handle VISA traffic, consult with your network administrator or network equipment providers.

Communicating with Devices over LAN

VISA supports LAN-gatewayed sessions. What this means is that you can communicate with configured LAN servers. The LAN server configuration is determined by the type of server present. The only action required by the user is to configure VISA for a **VISA LAN Client**. This configuration is done during the HP I/O Libraries configuration. See the *HP I/O Libraries Installation and Configuration Guide* for information on configuring a **VISA LAN Client**.

Addressing a Session

The same rules apply as when addressing a GPIB session. The only difference is that you use the VISA Interface Name provided during the I/O configuration that relates to the **VISA LAN Client**. The following illustrates addressing a GPIB device configured over the LAN:

`GPIB0::7::0` A GPIB device at primary address 7 and secondary address 0 on the GPIB interface. Note that this GPIB interface (GPIB0) happens to be configured as a **VISA LAN Client** in the HP I/O Libraries configuration.

The following is an example of opening a device 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);
```

See Chapter 4, "Programming with HP VISA," for more information on how to address device sessions.

LAN Session Example

The following C program example is the same example program as shown in Chapter 4, "Programming with HP VISA," only the address is modified to the GPIB device connected over LAN. This example opens a session with a GPIB device and sends a comma operator to send a comma-separated list. This example program is intended to show specific VISA functionality and does not include error trapping. Error trapping, however, is good programming practice and is recommended in your VISA applications. See "Trapping Errors" in Chapter 4, "Programming with HP VISA."

```
/*formatio.c
  This example program makes a multimeter measurement with a comma
  separated list passed with formatted I/O and prints the results.
  Note that you must change the device address. */

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

void main () {

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

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

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

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

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

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

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

Using Timeouts with LAN

The client/server architecture of the LAN software requires the use of two timeout values, one for the client and one for the server. The server's timeout value is specified by setting a VISA timeout via the `VI_ATTR_TMO_VALUE` attribute. The server will also adjust the requested value if infinity is requested. The client's timeout value is determined by the values set when you configure the `LAN Client` during the HP I/O Libraries configuration. See the *HP I/O Libraries Installation and Configuration Guide* for configuration information.

When the client sends an I/O request to the server, the timeout value determined by the values set with the `VI_ATTR_TMO_VALUE` attribute is passed with the request. The client may also adjust the value sent to the server if `VI_TMO_INFINITE` was specified. The server will use that timeout in performing the I/O operation, just as if that timeout value had been used on a local I/O operation. If the server's operation is not complete in the specified time, then the server will send a reply to the client which indicates that a timeout occurred, and the VISA call made by the application will return an error.

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

Default LAN Timeout Values

The `LAN Client` configuration specifies two timeout-related configuration values for the LAN software. These values are used by the software to calculate timeout values:

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

See the *HP I/O Libraries Installation and Configuration Guide* for information on setting these values.

The timeouts are adjusted via the following algorithm:

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

To change the defaults, do the following:

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

Application Terminations and Timeouts

If an application is killed either via Ctrl-C or the HP-UX `kill` command while in the middle of a VISA operation which is performed at the LAN server, the server will continue to try the operation until the server's timeout is reached. By default, the LAN server associated with an application using a timeout of infinity which is killed may not discover that the client is no longer running for 2 minutes. (If you are using a server other than the LAN server supported with this product, check that server's documentation for its default behavior.)

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

If long timeouts must be used, the server may be reset. An HP-UX server may be reset by logging into the server host and killing the running `siclland` daemon(s). Note that the latter procedure will affect all clients connected to the server. A Windows 95 or Windows NT server may be reset by typing Ctrl-C in the LAN Server window, and then restarting the server from the `HP I/O Libraries` program group. This procedure will also affect all clients connected to the server.

Using Signal Handling with LAN

SIGIO Signals

VISA uses SIGIO for SRQs on LAN interfaces on HP-UX. The VISA LAN client installs a signal handler to catch SIGIO signals. To enable sharing of SIGIO signals with other portions of an application, the VISA LAN SIGIO signal handler remembers the address of any previously installed SIGIO handler, and calls this handler after processing a SIGIO signal itself. If your application installs a SIGIO handler, it should also remember the address of a previously installed handler and call it before completing.

The signal number used with LAN (SIGIO) can *not* be changed.

HP VISA Function Support with LAN

A LAN session to a remote interface provides the same VISA function support as if the interface was local, with the following exceptions or qualifications.

All VXI specific functions are *not* supported over LAN.

GPIB Sessions and Service Requests over LAN

If multiple devices assert SRQs at roughly the same time causing the SRQ line to stay asserted, even after all devices have been polled using **viReadSTB**, then subsequent service requests from devices may be lost since the SRQ handler(s) will not be invoked again until the line is cleared. For SRQs to be reliably delivered, an SRQ handler must not exit without first clearing the SRQ line. However, VISA does not provided a way to check the SRQ line.

One way to ensure reliable delivery of SRQs is to service all devices from one handler, disabling all devices from sending additional SRQs at the top of the handler. See the following:

```
disable all devices from requesting service
serial_poll (device1)
if (needs_service) service_device1
serial_poll (device2)
if (needs_service) service_device2
.
.
enable all devices to send service requests
```

Even if the different sessions are in different processes, it is important to stay in the SRQ handler until the SRQ line is released. However, the only way to ensure true independence of multiple GPIB processes is to use multiple GPIB interfaces.

Another way in which this situation can be avoided is if a VISA LAN client is configured to use the SICL LAN protocol and the LAN server is a Windows 95, Windows NT, or HP-UX 10.x system running the LAN server that is shipped as part of this product. This method is handled transparently, just as for other interfaces.

7

HP VISA Language Reference

HP VISA Language Reference

This chapter describes each function in the VISA library for the Windows and HP-UX programming environments. The VISA functions are provided in alphabetical order in this chapter for easy reference.

The VISA functions can be grouped according to the types of functions performed, as shown in the following table. Note that the `OUT` parameters are identified by the type definition. In other words, all `OUT` parameters are defined with a pointer type: `ViPUInt16`, `ViPRsrc`, and so forth.

NOTE

The data types for the VISA function parameters (for example, `ViSession`, `ViEventType`, and so forth) are defined in the VISA declarations file. They are also explained in Appendix D, "HP VISA Type Definitions," in this manual.

VISA Functions

Operation	Function(Type *Parameter1*, Type *Parameter2*, ...);
Resource Management:	
Open Default Resource Manager Session	viOpenDefaultRM(ViPSession *sesn*);
Lifecycle:	
Open Session	viOpen(ViSession *sesn*, ViRsrc *rsrcName*, ViAccessMode *accessMode*, ViUInt32 *timeout*, ViPSession *vi*);
Close Session	viClose(ViSession/ViEvent/ViFindList *vi*);
Characteristic Control:	
Get Attribute	viGetAttribute(ViSession/ViEvent/ViFindList *vi*, ViAttr *attribute*, ViPAttrState *attrState*);
Set Attribute	viSetAttribute(ViSession/ViEvent/ViFindList *vi*, ViAttr *attribute*, ViAttrState *attrState*);
Get Status Code Description	viStatusDesc(ViSession/ViEvent/ViFindList *vi*, ViStatus *status*, ViPString *desc*);
Asynchronous Operation Control:	
Terminate Asynchronous Operation	viTerminate(ViSession *vi*, ViUInt16 *degree*, ViJobId *jobId*);
Access Control:	
Lock Resource	viLock(ViSession *vi*, ViAccessMode *lockType*, ViUInt32 *timeout*, ViKeyId *requestedKey*, ViPKeyId *accessKey*);
Unlock Resource	viUnlock(ViSession *vi*);
Event Handling:	
Enable Event	viEnableEvent(ViSession *vi*, ViEventType *eventType*, ViUInt16 *mechanism*, ViEventFilter *context*);
Disable Event	viDisableEvent(ViSession *vi*, ViEventType *eventType*, ViUInt16 *mechanism*);
Discard Events	viDiscardEvents(ViSession *vi*, ViEventType *eventType*, ViUInt16 *mechanism*);

VISA Functions (continued)

Operation	Function(Type *Parameter1*, Type *Parameter2*, ...);
Event Handling (continued):	
Wait on Event	viWaitOnEvent(ViSession *vi*, ViEventType *inEventType*, ViUInt32 *timeout*, ViPEventType *outEventType*, ViPEvent *outContext*);
Install Handler	viInstallHandler(ViSession *vi*, ViEventType *eventType*, ViHndlr *handler*, ViAddr *userHandle*);
Uninstall Handler	viUninstallHandler(ViSession *vi*, ViEventType *eventType*, ViHndlr *handler*, ViAddr *userHandle*);
Event Handler Prototype	viEventHandler(ViSession *vi*, ViEventType *eventType*, ViEvent *context*, ViAddr *userHandle*);
Searching:	
Find Device	viFindRsrc(ViSession *sesn*, ViString *expr*, ViPFindList *findList*, ViPUInt32 *retcnt*, ViPRsrc *instrDesc*);
Find Next Device	viFindNext(ViFindList *findList*, ViPRsrc *instrDesc*);
Basic I/O:	
Read Data from Device	viRead(ViSession *vi*, ViPBuf *buf*, ViUInt32 *count*, ViPUInt32 *retCount*);
Read Data Asynchronously from Device	viReadAsync(ViSession *vi*, ViPBuf *buf*, ViUInt32 *count*, ViPJobId *jobId*);
Write Data to Device	viWrite(ViSession *vi*, ViBuf *buf*, ViUInt32 *count*, ViPUInt32 *retCount*);
Write Data Asynchronously to Device	viWriteAsync(ViSession *vi*, ViBuf *buf*, ViUInt32 *count*, ViPJobId *jobId*);
Assert Software/Hardware Trigger	viAssertTrigger(ViSession *vi*, ViUInt16 *protocol*);
Read Status Byte	viReadSTB(ViSession *vi*, ViPUInt16 *status*);
Clear a Device	viClear(ViSession *vi*);

VISA Functions (continued)

Operation	Function(Type *Parameter1*, Type *Parameter2*, ...);
Formatted I/O:	
Set Size of Buffer	viSetBuf(ViSession *vi*, ViUInt16 *mask*, ViUInt32 *size*);
Flush Read and Write Buffers	viFlush(ViSession *vi*, ViUInt16 *mask*);
Convert, Format, and Send Parameters	viPrintf(ViSession *vi*, ViString *writeFmt*, *arg1*, *arg2*, ...);
Convert, Format, and Send Parameters	viVPrintf(ViSession *vi*, ViString *writeFmt*, ViVAList *params*);
Read, Convert, Format, and Store Data	viScanf(ViSession *vi*, ViString *readFmt*, *arg1*, *arg2*, ...);
Read, Convert, Format, and Store Data	viVScanf(ViSession *vi*, ViString *readFmt*, ViVAList *params*);
Write and Read Formatted Data	viQueryf(ViSession *vi*, ViString *writeFmt*, ViString *readFmt*, *arg1*, *arg2*, ...);
Write and Read Formatted Data	viVQueryf(ViSession *vi*, ViString *writeFmt*, ViString *readFmt*, ViVAList *params*);
Memory I/O:	
Read 8-bit Value from Memory Space	viIn8(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViPUInt8 *val8*);
Read 16-bit Value from Memory Space	viIn16(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViPUInt16 *val16*);
Read 32-bit Value from Memory Space	viIn32(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViPUInt32 *val32*);
Write 8-bit Value to Memory Space	viOut8(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViUInt8 *val8*);
Write 16-bit Value to Memory Space	viOut16(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViUInt16 *val16*);
Write 32-bit Value to Memory Space	viOut32(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViUInt32 *val32*);

VISA Functions (continued)

Operation	Function(Type *Parameter1*, Type *Parameter2*, ...);
Memory I/O (continued):	
Move 8-bit Value from Device Memory to Local Memory	viMoveIn8(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt8 *buf8*);
Move 16-bit Value from Device Memory to Local Memory	viMoveIn16(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt16 *buf16*);
Move 32-bit Value from Device Memory to Local Memory	viMoveIn32(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt32 *buf32*);
Move 8-bit Value from Local Memory to Device Memory	viMoveOut8(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt8 *buf8*);
Move 16-bit Value from Local Memory to Device Memory	viMoveOut16(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt16 *buf16*);
Move 32-bit Value from Local Memory to Device Memory	viMoveOut32(ViSession *vi*, ViUInt16 *space*, ViBusAddress *offset*, ViBusSize *length*, ViAUInt32 *buf32*);
Map Memory Space	viMapAddress(ViSession *vi*, ViUInt16 *mapSpace*, ViBusAddress *mapBase*, ViBusSize *mapSize*, ViBoolean *access*, ViAddr *suggested*, ViPAddr *address*);
Unmap Memory Space	viUnmapAddress(ViSession *vi*);
Read 8-bit Value from Address	viPeek8(ViSession *vi*, ViAddr *addr*, ViPUInt8 *val8*);
Read 16-bit Value from Address	viPeek16(ViSession *vi*, ViAddr *addr*, ViPUInt16 *val16*);
Read 32-bit Value from Address	viPeek32(ViSession *vi*, ViAddr *addr*, ViPUInt32 *val32*);
Write 8-bit Value to Address	viPoke8(ViSession *vi*, ViAddr *addr*, ViUInt8 *val8*);
Write 16-bit Value to Address	viPoke16(ViSession *vi*, ViAddr *addr*, ViUInt16 *val16*);
Write 32-bit Value to Address	viPoke32(ViSession *vi*, ViAddr *addr*, ViUInt32 *val32*);
Shared Memory:	
Allocate Memory	viMemAlloc(ViSession *vi*, ViBusSize *size*, ViPBusAddress *offset*);
Free Memory Previously Allocated	viMemFree(ViSession *vi*, ViBusAddress *offset*);

The following sections explain each of the VISA functions in alphabetical order.

viAssertTrigger

Syntax

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

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

Description

This function asserts 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`, then the device is sent the Word Serial Trigger command. For a VXI device, if `VI_ATTR_TRIG_ID` is any other value, a hardware trigger is sent on the line corresponding to the value of that attribute.

For GPIB 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`.

Parameters

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

Return Values

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

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

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

viClear

Syntax

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

Description

This function performs an IEEE 488.1-style clear of the device. VXI uses the Word Serial Clear command, and GPIB uses the Selective Device Clear command.

Parameters

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

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

viClose

Syntax

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

Description

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

> **NOTE**
>
> The **viClose** function should not be called from within an event handler.

Parameters

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Session closed successfully.

Error Code	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_CLOSING_FAILED`	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

See Also

"viOpen", "viFindRsrc", "viWaitOnEvent", "viEventHandler"

viDisableEvent

Syntax

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

Description

This function disables servicing of an event identified by the *eventType* parameter for the mechanisms specified in the *mechanism* parameter. Specifying `VI_ALL_ENABLED_EVENTS` for the *eventType* parameter allows a session to stop receiving all events. The session can stop receiving queued events by specifying `VI_QUEUE`. Applications can stop receiving callback events by specifying either `VI_HNDLR` or `VI_SUSPEND_HNDLR`. Specifying `VI_ALL_MECH` disables both the queuing and callback mechanisms.

Parameters

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

Special Values for *eventType* Parameter

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

The following events can be disabled:

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

Special Values for *mechanism* Parameter

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

Return Values

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

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

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

See Also

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

viDiscardEvents

Syntax

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

Description

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

Parameters

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

Special Values for *eventType* Parameter

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

The following events can be discarded:

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

Special Values for *mechanism* Parameter

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

Return Values

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

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

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

See Also

"viEnableEvent", "viWaitOnEvent", "viInstallHandler"

viEnableEvent

Syntax

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

Description

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

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
eventType	IN	**ViEventType**	Logical event identifier. (See the following tables.)
mechanism	IN	**ViUInt16**	Specifies event handling mechanisms to be enabled. The queuing mechanism is enabled by specifying **VI_QUEUE**, and the callback mechanism is enabled by specifying **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	Action Description
VI_ALL_ENABLED_EVENTS	Switch all events that were previously enabled to the callback mechanism specified in the mechanism parameter.

The following events can be enabled:

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

Special Values for *mechanism* Parameter

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

NOTE

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

Return Values

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

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

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

See Also

See the handler prototype, "viEventHandler", for its parameter description, and "viDisableEvent". Also refer to the "viInstallHandler" and "viUninstallHandler" descriptions for information about installing and uninstalling event handlers.

viEventHandler

Syntax

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

Description

This is a prototype for a function, which you define. The function you define is called whenever a session receives an event and is enabled for handling events in the `VI_HNDLR` mode. The handler services the event and returns `VI_SUCCESS` on completion.

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.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
eventType	IN	`ViEventType`	Logical event identifier. (See the following table.)
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.

The following table lists the events and the associated read only attributes that can be read to get event information on a specific event:

Event Name	Attributes	Data Type	Values
VI_EVENT_SERVICE_REQ	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_REQ
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE VI_ATTR_SIGP_STATUS_ID	ViEventType ViUInt16	VI_EVENT_VXI_SIGP 0 to FFFFh
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	ViEventType ViStatus ViJobId ViBuf ViUInt32	VI_EVENT_IO_COMPLETION N/A N/A N/A 0 to FFFFFFFFh

Use the VISA **viReadSTB** function to read the status byte of the service request.

Return Values

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

NOTE

Return values are not used in VISA 1.0, but will be significant in future versions of VISA. Therefore, you should always return **VI_SUCCESS** from an event handler.

Completion Code	Description
VI_SUCCESS	Event handled successfully.

See Also

Refer to the "Using Events and Handlers" section of Chapter 4, "Programming with HP VISA," for more information on event handling and exception handling.

viFindNext

Syntax

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

Description

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

Parameters

Name	Direction	Type	Description
findList	IN	`ViFindList`	Describes a find list. This parameter must be created by `viFindRsrc`.
instrDesc	OUT	`ViPRsrc`	Returns a string identifying the location of a device. Strings can then be passed to `viOpen` to establish a session to the given 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 Code	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 *findList* does not support this function.
`VI_ERROR_RSRC_NFOUND`	There are no more matches.

See Also

"viFindRsrc"

viFindRsrc

Syntax

```
viFindRsrc(ViSession sesn, ViString expr, ViPFindList findList,
  ViPUInt32 retcnt, ViPRsrc instrDesc);
```

Description

This function queries a VISA system to locate the devices associated with a specified interface. This function matches the value specified in the *expr* parameter with the devices available for a particular interface. On successful completion, it returns the first device found in the list and returns a count to indicate if there were more devices found that match the value specified in the *expr* parameter.

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

Parameters

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

Description String for *expr* Parameter

Interface	Expression
GPIB	`GPIB[0-9]*::?*INSTR`
VXI	`VXI?*INSTR`
GPIB-VXI	`GPIB-VXI?*INSTR`
GPIB and GPIB-VXI	`GPIB?*INSTR`
All VXI	`?*VXI[0-9]*::?*INSTR`
ASRL	`ASRL[0-9]*::?*INSTR`
All	`?*INSTR`

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

See Also

"viFindNext", "viClose"

viFlush

Syntax

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

Description

This function manually flushes the read and write buffers associated with formatted I/O functions.

Parameters

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

Values for *mask* Parameter

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

NOTE

It is possible to combine any of these read flags with a write flag (and vice versa) by ORing the flags. However, combining two read flags or two write flags in the same call to **viFlush** is illegal.

NOTE

In this implementation, it is not possible to discard the ASRL in and out buffers separately. `VI_ASRL_IN_BUF_DISCARD` and `VI_ASRL_OUT_BUF_DISCARD` must always be set together. If only one is set, `VI_ERROR_INV_MASK` is returned.

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

See Also

"viSetBuf"

viGetAttribute

Syntax

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

Description

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

Parameters

Name	Direction	Type	Description

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

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

NOTE

The pointer passed to `viGetAttribute` must point to the exact type required for that attribute, `ViUInt16`, `ViInt32`, and so forth. 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 Code	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"

viIn8, viIn16, and viIn32

Syntax

```
viIn8(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViPUInt8 val8);

viIn16(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViPUInt16 val16);

viIn32(ViSession vi, ViUInt16 space, ViBusAddress offset,
  ViPUInt32 val32);
```

Description

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

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
space	IN	**ViUInt16**	Specifies the address space. (See the following table.)
offset	IN	**ViBusAddress**	Offset (in bytes) of the memory to read from.
val8, val16, or *val32*	OUT	**ViPUInt8**, **ViPUInt16**, or **ViPUInt32**	Data read from bus (8-bits for **viIn8**, 16-bits for **viIn16**, and 32-bits for **viIn32**).

Values for *space* Parameter

Value	Description
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.

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

See Also

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

viInstallHandler

Syntax

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

Description

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

Parameters

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

The following events can be enabled:

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Event handler installed successfully.

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

See Also

"viEventHandler"

viLock

Syntax

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

NOTE

The `viLock` function is *not* supported with 16-bit VISA on Windows 95.

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 the *accessKey* parameters apply only to shared locks. These parameters are not applicable when using the lock type `VI_EXCLUSIVE_LOCK`. In this case, *requestedKey* and *accessKey* should be set to `VI_NULL`. VISA allows user applications to specify a key to be used for lock sharing through the use of the *requestedKey* parameter. Alternatively, a user application can pass `VI_NULL` for the *requestedKey* parameter when obtaining a shared lock, in which case VISA will generate a unique access key and return it through the *accessKey* parameter. If a user application does specify a *requestedKey* value, VISA will try to use this value for the *accessKey*. As long as the resource is not locked, VISA will use the *requestedKey* as the access key and grant the lock. When the operation succeeds, the *requestedKey* will be copied into the user buffer referred to by the *accessKey* parameter.

The session that gained a shared lock can pass the *accessKey* to other sessions for the purpose of sharing the lock. The session wanting to join the group of sessions sharing the lock can use the key as an input value to the *requestedKey* parameter. VISA will add the session to the list of sessions sharing the lock, as long as the *requestedKey* value matches the *accessKey* value for the particular resource. The session obtaining a shared lock in this manner will then have the same access privileges as the original session that obtained the lock.

It is also possible to obtain nested locks through this function. To acquire nested locks, invoke the **viLock** function with the same lock type as the previous invocation of this function. For each session, **viLock** and **viUnlock** share a lock count, which is initialized to 0. Each invocation of **viLock** for the same session (and for the same *lockType*) increases the lock count. In the case of a shared lock, it returns with the same *accessKey* every time. When a session locks the resource a multiple number of times, it is necessary to invoke the **viUnlock** function an equal number of times in order to unlock the resource. That is, the lock count increments for each invocation of **viLock**, and decrements for each invocation of **viUnlock**. A resource is actually unlocked only when the lock count is 0.

NOTE

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

NOTE

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

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

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
lockType	IN	`ViAccessMode`	Specifies the type of lock requested, which can be either `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.
requestedKey	IN	`ViKeyId`	This parameter is not used and should be set to `VI_NULL` when `lockType` is `VI_EXCLUSIVE_LOCK` (exclusive lock). When trying to lock the resource as `VI_SHARED_LOCK` (shared lock), a session can either set it to `VI_NULL` so that VISA generates an *accessKey* for the session, or the session can suggest an *accessKey* to use for the shared lock. Refer to the previous "Description" subsection for more details.
accessKey	OUT	`ViPKeyId`	This parameter should be set to `VI_NULL` when *lockType* is `VI_EXCLUSIVE_LOCK` (exclusive lock). When trying to lock the resource as `VI_SHARED_LOCK` (shared lock), the resource returns a unique access key for the lock if the operation succeeds. This *accessKey* can then be passed to other sessions to share the lock.

Return Values

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

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

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

See Also

"viUnlock". For more information on locking, see the "Using Locks" section of Chapter 4, "Programming with HP VISA."

viMapAddress

Syntax

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

Description

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

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.

Parameters

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

Values for *mapSpace* Parameter

Value	Description
`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.

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 Code	Description
VI_ERROR_INV_SESSION VI_ERROR_INV_OBJECT	The given session or object reference is invalid (both are the same value).
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SPACE	Invalid *mapSpace* specified.
VI_ERROR_INV_OFFSET	Invalid *offset* specified.
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.
VI_ERROR_TMO	**viMapAddress** could not acquire resource or perform mapping before the timer expired.
VI_ERROR_INV_SIZE	Invalid size of window specified.
VI_ERROR_ALLOC	Unable to allocate window of at least the requested size.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.

See Also

"viUnmapAddress"

viMemAlloc

Syntax

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

Description

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

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
size	IN	`ViBusSize`	Specifies the size of the allocation.
offset	OUT	`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`	The operation completed successfully.

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

See Also

"viMemFree"

viMemFree

Syntax

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

Description

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

Parameters

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

Return Values

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

Completion Code	Description
VI_SUCCESS	The operation completed successfully.

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

See Also

"viMemAlloc"

viMoveIn8, viMoveIn16, and viMoveIn32

Syntax

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

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

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

Description

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

NOTE

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

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
space	IN	`ViUInt16`	Specifies the address space. (See the following table.)
offset	IN	`ViBusAddress`	Offset (in bytes) of the memory to read from.
length	IN	`ViBusSize`	Number of elements to transfer, where the data width of the elements to transfer is 8-bits for `viMoveIn8`, 16-bits for `viMoveIn16`, or 32-bits for `viMoveIn32`.
buf8, buf16, or *buf32*	OUT	`ViAUInt8`, `ViAUInt16`, or `ViAUInt32`	Data read from bus (8-bits for `viMoveIn8`, 16-bits for `viMoveIn16`, and 32-bits for `viMoveIn32`).

Values for *space* Parameter

Value	Description
`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.

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

See Also "viMoveOut8, viMoveOut16, and viMoveOut32", "viIn8, viIn16, and viIn32"

viMoveOut8, viMoveOut16, and viMoveOut32

Syntax

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

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

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

Description

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

NOTE

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).

Parameters

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

Values for *space* Parameter

Value	Description
`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.

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

See Also

"viMoveIn8, viMoveIn16, and viMoveIn32", "viOut8, viOut16, and viOut32"

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.

Parameters

Name	Direction	Type	Description
sesn	IN	`ViSession`	Resource Manager session (should always be the Default Resource Manager for VISA returned from `viOpenDefaultRM`).
rsrcName	IN	`ViRsrc`	Unique symbolic name of a resource. (See the following tables.)
accessMode	IN	`ViAccessMode`	`VI_NULL` (Not used for VISA 1.0.)
timeout	IN	`ViUInt32`	`VI_NULL` (Not used for VISA 1.0.)
vi	OUT	`ViPSession`	Unique logical identifier reference to a session.

Address String Grammar for *rsrcName* Parameter

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

Examples of Address Strings for *rsrcName* Parameter

Address String	Description
`VXI0::1::INSTR`	A VXI device at logical address `1` in VXI interface `VXI0`.
`GPIB-VXI::24::INSTR`	A VXI device at logical address `24` 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`.

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 opened successfully.

Error Code	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_NSUP_OPER`	The given *sesn* does not support this function. For VISA, this function is supported only by the Default Resource Manager session.
`VI_ERROR_INV_RSRC_NAME`	Invalid resource reference specified. Parsing error.
`VI_ERROR_INV_ACC_MODE`	Invalid access mode.
`VI_ERROR_RSRC_NFOUND`	Insufficient location information or resource not present in the system.
`VI_ERROR_ALLOC`	Insufficient system resources to open a session.

See Also

"viClose"

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 that you will be using need to be connected and in working condition prior to the first VISA function call (`viOpenDefaultRM`). The system is configured only on the *first* `viOpenDefaultRM` per process. Therefore, if `viOpenDefaultRM` is 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 in working condition.

Parameters

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

Return Values

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

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

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

See Also

"viOpen", "viFindRsrc", "viClose"

viOut8, viOut16, and viOut32

Syntax

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

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

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

Description

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

Parameters

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

Values for *space* Parameter

Value	Description
`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.

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

See Also

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

viPeek8, viPeek16, and viPeek32

Syntax

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

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

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

Description

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

Parameters

Name	Direction	Type	Description
vi	IN	**ViSession**	Unique logical identifier to a session.
addr	IN	**ViAddr**	Specifies the source address to read the value.
val8, val16, or *val32*	OUT	**ViPUInt8**, **ViPUInt16**, or **ViPUInt32**	Data read from bus (8-bits for **viPeek8**, 16-bits for **viPeek16**, and 32-bits for **viPeek32**).

NOTE

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

Return Values

None.

See Also

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

viPoke8, viPoke16, and viPoke32

Syntax

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

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

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

Description

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

Parameters

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

NOTE

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

Return Values

None.

See Also

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

viPrintf

Syntax

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

Description

This function converts, formats, and sends the parameters *arg1, arg2, ...* to the device as specified by the format string. Before sending the data, the function formats the *arg* characters in the parameter list as specified in the *writeFmt* string.

You should not use the `viWrite` and `viPrintf` functions in the same session.

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

Parameters

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

Special Formatting Characters

The special formatting characters and what they send to the device are:

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

Format Specifiers

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

% [*modifiers*] *conversion character*

where *conversion character* specifies which data type the argument is represented in. The *modifiers* are optional codes that describe the target data.

In the following tables, a `d` conversion character refers to all conversion codes of type integer (`d, i, o, u, x, X`), unless specified as **%**`d` only. Similarly, an `f` conversion character refers to all conversion codes of type float (`f, e, E, g, G`), unless specified as **%**`f` only.

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

ANSI C Standard Modifiers

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

Enhanced Modifiers to ANSI C Standards

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

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

Standard ANSI C Conversion Characters

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

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

`d` Argument type: An integer.

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

`f` Argument type: A floating point number.

Modifier	Interpretation
Default functionality	Print a floating point number in NR2 format (a number with at least one digit after the decimal point).
@1	Print an integer in NR1 format. The number is truncated.
@3	Print a floating point number in NR3 format (scientific notation). *Precision* can also be specified.
field width	Minimum field width of the output number. Any of the six IEEE 488.2 modifiers can also be specified with *field width*.
Length modifier `l`	*arg* is a double float.
Length modifier `L`	*arg* is a long double.
, *array size*	*arg* points to an array of floats (or doubles or long doubles), depending on the length modifier) of size *array size*. The elements of this array are separated by *array size* − 1 commas and output in the specified format.

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

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

Flag or Modifier	Interpretation
Default functionality	The data block is sent as an IEEE 488.2 <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>. A count (long) must appear as a flag that specifies the number of bytes in the block. A *field width* or *precision* modifier is not allowed with this conversion character.
* (asterisk)	An asterisk may be present instead of the count. In such a case, two *args* are used, the first of which is a count of the number of elements in the data block. The second *arg* is a reference to the data block. The size of an element is determined by the optional length modifier (see below), the default being byte width.
Length modifier `h`	The data block is assumed to be an array of unsigned short integers (16-bit word). 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.
Length modifier `Z`	The data block is assumed to be an array of doubles. The count corresponds to the number of double floats (64-bits). The numbers are represented in IEEE 754 (big endian) format if native computer representation is different.

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

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

See Also

"viVPrintf"

viQueryf

Syntax

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

Description

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

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

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

Parameters

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

Return Values

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

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

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

See Also

"viPrintf", "viScanf", "viVQueryf"

viRead

Syntax

```
viRead(ViSession vi, 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.

> **NOTE**
>
> You must set specific attributes to make the read terminate under specific conditions. See Appendix B, "HP VISA Attributes."

Parameters

Name	Direction	Type	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.

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

See Also

"viWrite"

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.

Parameters

Name	Direction	Type	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.

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`	Asynchronous read operation successfully queued.
`VI_SUCCESS_SYNC`	Read operation performed synchronously.

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

See Also

"viRead", "viTerminate", "viWrite", "viWriteAsync"

viReadSTB

Syntax

```
viReadSTB(ViSession vi, ViPUInt16 status);
```

Description

This function reads a status byte of the service request 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. If the status information is only one byte long, the most significant byte is returned with the zero value. If the service requester does not respond in the actual timeout period, `VI_ERROR_TMO` is returned.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to the session.
status	OUT	`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.

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

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

viScanf

Syntax

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

Description

This function receives data from a device, formats it by using the format string, and stores the data in the *arg* parameter list. The format string can have format specifier sequences, white space characters, and ordinary characters. The white characters (blank, vertical tabs, horizontal tabs, form feeds, new line/linefeed, and carriage return) are ignored except in the case of `%c` and `%[]`. All other ordinary characters except `%` should match the next character read from the device.

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

`%`[*modifiers*]*conversion character*

where the optional *modifier* describes the data format, while *conversion character* indicates the nature of data (data type). One and only one *conversion character* should be performed at the specifier sequence. A format specification directs the conversion to the next input *arg*. The results of the conversion are placed in the variable that the corresponding argument points to, unless the asterisk (`*`) assignment-suppressing character is given. In such a case, no *arg* is used, and the results are ignored.

The `viScanf` function accepts input until an END indicator is read or all the format specifiers in the *readFmt* string are satisfied. It also terminates if the format string character does not match the incoming character. Thus, detecting an END indicator before the *readFmt* string is fully consumed will result in ignoring the rest of the format string. Also, if some data remains in the buffer after all format specifiers in the *readFmt* string are satisfied, the data will be kept in the buffer and will be used by the next `viScanf` function.

Parameters

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

The following two tables describe optional modifiers that can be used in a format specifier sequence.

ANSI C Standard Modifiers

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

Enhanced Modifiers to ANSI C Standards

Modifier	Supported with Conversion Character	Description
A comma (`,`) followed by an integer *n*, where *n* represents the array size.	`%d` and `%f` only	The corresponding argument is interpreted as a reference to the first element of an array of size *n*. The first *n* elements of this list are printed in the format specified by the conversion character. A number sign (`#`) may be present after the `,` modifier, in which case an extra *arg* is used. This *arg* must be an integer representing the array size of the given type.

Conversion Characters

ANSI C Conversion Characters

`c` Argument type: A reference to a character.

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

NOTE

White space in the device input stream is not ignored when using `%c`.

`d` Argument type: A reference to an integer.

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

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

Flags or Modifiers	Interpretation
Default functionality	Characters are read from the device until an entire number is read. The number read must be in either IEEE 488.2 formats: <DECIMAL NUMERIC PROGRAM DATA> (NRf), or <NON-DECIMAL NUMERIC PROGRAM DATA> (**#H**, **#Q**, and **#B**).
field width	The input number will be stored in a field at least this wide.
Length modifier `l`	*arg* is a reference to a double floating point number.
Length modifier `L`	*arg* is a reference to a long double number.
, *array size*	*arg* points to an array of floats (or doubles or long doubles, depending on the length modifier) of size *array size*. The elements of this array should be separated by commas. Elements will be read until either *array size* number of elements are consumed or they are no longer separated by commas.

s Argument type: A reference to a string.

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

Enhanced Conversion Characters

`b` Argument type: A reference to a data array.

Flags or Modifiers	Interpretation
Default functionality	The data must be in IEEE 488.2 <ARBITRARY BLOCK PROGRAM DATA> format. The format specifier sequence should have a flag describing the *field width*, 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 *field width* contains a **#** sign, two arguments are used. The first argument read gives the maximum size of the array. The second one should be a reference to an array. Also in this case, the actual number of elements read is stored back in the first argument. In absence of length modifiers, the data is assumed to be of byte-size elements. In some cases, data might be read until an END indicator is read.
Length modifier **h**	The array is assumed to be an array of 16-bit words, and count refers to the number of words. The data read from the interface is assumed to be in IEEE 488.2 (big endian) byte ordering. It will be byte swapped and padded as appropriate to the native computer format.
Length modifier **l**	The array is assumed to be a block of 32-bit longwords rather than bytes, and count refers to the number of longwords. The data read from the interface is assumed to be in IEEE 488.2 (big endian) byte ordering. It will be byte swapped and padded as appropriate to the native computer format.
Length modifier **z**	The data block is assumed to be a reference to an array of floats, and count refers to the number of floating point numbers. The data block received from the device is an array of 32-bit IEEE 754 format floating point numbers.
Length modifier **Z**	The data block is assumed to be a reference to an array of doubles, and the count refers to the number of floating point numbers. The data block received from the device is an array of 64-bit IEEE 754 format floating point numbers.

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

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

T Argument type: A reference to a string.

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

Return Values

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

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

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

See Also

"viVScanf"

viSetAttribute

Syntax

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

Description

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

Parameters

Name	Direction	Type	Description

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

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

See Also

"viGetAttribute". Also refer to Appendix B, "HP VISA Attributes," for a list of attributes and attribute values. Chapter 4, "Programming with HP VISA," provides detailed descriptions of the VISA attributes.

viSetBuf

Syntax

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

Description

This function sets the size of the read and/or write buffer for formatted I/O and/or serial communication. The *mask* parameter specifies whether the buffer is a read or write buffer. 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_ASRL_IN_BUF	Serial communication receive buffer.
VI_ASRL_OUT_BUF	Serial communication transmit buffer.

Parameters

Name	Direction	Type	Description
vi	IN	ViSession	Unique logical identifier to a session.
mask	IN	ViUInt16	Specifies the type of buffer. (See previous table.)
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 Code	Description
VI_SUCCESS	Buffer size set successfully.
VI_WARN_NSUP_BUF	The specified buffer is not supported.

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

viStatusDesc

Syntax

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

Description

This function returns a user-readable string which describes the status code passed to the function.

Parameters

Name	Direction	Type	Description

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

status	IN	`ViStatus`	Status code to interpret.
desc	OUT	`ViPString`	The user-readable string interpretation of the status code passed to the function.

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`	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);
```

Description

This function requests a VISA session to terminate normal execution of an asynchronous operation.

Parameters

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Request serviced successfully.

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

See Also

"viReadAsync", "viWriteAsync"

viUninstallHandler

Syntax

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

Description

This function allows applications to uninstall handlers for events on sessions. Applications should also specify the value in the *userHandle* parameter that was passed to `viInstallHandler` while installing the handler. VISA identifies handlers uniquely using the *handler* reference and this value. All the handlers, for which the *handler* reference and the value matches, are uninstalled.

Parameters

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

The following events are valid:

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

Special Values for *handler* Parameter

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

Return Values

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

Completion Code	Description
VI_SUCCESS	Event handler successfully uninstalled.

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

See Also

See the handler prototype, "viEventHandler", for its parameter description. Also refer to the "viEnableEvent" description for information about enabling different event handling mechanisms. Refer to individual event descriptions for context definitions.

viUnlock

Syntax

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

> **NOTE**
>
> The `viUnlock` function is *not* supported with 16-bit VISA on Windows 95.

Description

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

Parameters

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

Return Values

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

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

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

See Also

"viLock". For more information on locking, see the "Using Locks" section of Chapter 4, "Programming with HP VISA."

viUnmapAddress

Syntax

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

Description

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

Parameters

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Operation completed successfully.

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

See Also

"viMapAddress"

viVPrintf

Syntax

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

Description

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

Parameters

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

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Parameters were successfully formatted.

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

See Also

"viPrintf"

viVQueryf

Syntax

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

Description

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

Parameters

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

Return Values

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

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

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

See Also

"viVPrintf", "viVScanf", "viQueryf"

viVScanf

Syntax

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

Description

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

Parameters

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

Return Values

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

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

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

See Also

"viScanf"

viWaitOnEvent

Syntax

```
viWaitOnEvent(ViSession vi, ViEventType inEventType,
  ViUInt32 timeout, ViPEventType outEventType, ViPEvent 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*. Refer to 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.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
inEventType	IN	`ViEventType`	Logical identifier of the event(s) to wait for.
timeout	IN	`ViUInt32`	Absolute time period in time units that the resource shall wait for a specified event to occur before returning the time elapsed error. The time unit is in milliseconds.
outEventType	OUT	`ViPEventType`	Logical identifier of the event actually received.
outContext	OUT	`ViPEvent`	A handle specifying the unique occurrence of an event.

NOTE

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

The following table lists the events and the associated read only attributes that can be read using `viGetAttribute` to get event information on a specific event:

Event Name	Attributes	Data Type	Values
VI_EVENT_SERVICE_REQ	VI_ATTR_EVENT_TYPE	ViEventType	VI_EVENT_SERVICE_REQ
VI_EVENT_VXI_SIGP	VI_ATTR_EVENT_TYPE VI_ATTR_SIGP_STATUS_ID	ViEventType ViUInt16	VI_EVENT_VXI_SIGP 0 to FFFFh
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	ViEventType ViStatus ViJobId ViBuf ViUInt32	VI_EVENT_IO_COMPLETION N/A N/A N/A 0 to FFFFFFFFh

Use the VISA `viReadSTB` function to read the status byte of the service request.

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

Error Code	Description
`VI_ERROR_INV_SESSION` `VI_ERROR_INV_OBJECT`	The given session or object reference is invalid (both are the same value).
`VI_ERROR_INV_EVENT`	Specified event type is not supported by the resource.
`VI_ERROR_TMO`	Specified event did not occur within the specified time period.

See Also

Refer to the "Using Events and Handlers" section in Chapter 4, "Programming with HP VISA," 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.

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
buf	IN	`ViBuf`	Represents the location of a data block to be sent to device.
count	IN	`ViUInt32`	Specifies number of bytes to be written.
retCount	OUT	`ViPUInt32`	Represents the location of an integer that will be set to the number of bytes actually transferred.

Return Values

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

Completion Code	Description
`VI_SUCCESS`	Transfer completed.

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

See Also

"viRead"

viWriteAsync

Syntax

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

Description

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

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

Parameters

Name	Direction	Type	Description
vi	IN	`ViSession`	Unique logical identifier to a session.
buf	IN	`ViBuf`	Represents the location of a data block to be sent to the device.
count	IN	`ViUInt32`	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.

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`	Asynchronous write operation successfully queued.
`VI_SUCCESS_SYNC`	Write operation performed synchronously.

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

See Also "viRead", "viTerminate", "viWrite", "viReadAsync"

A

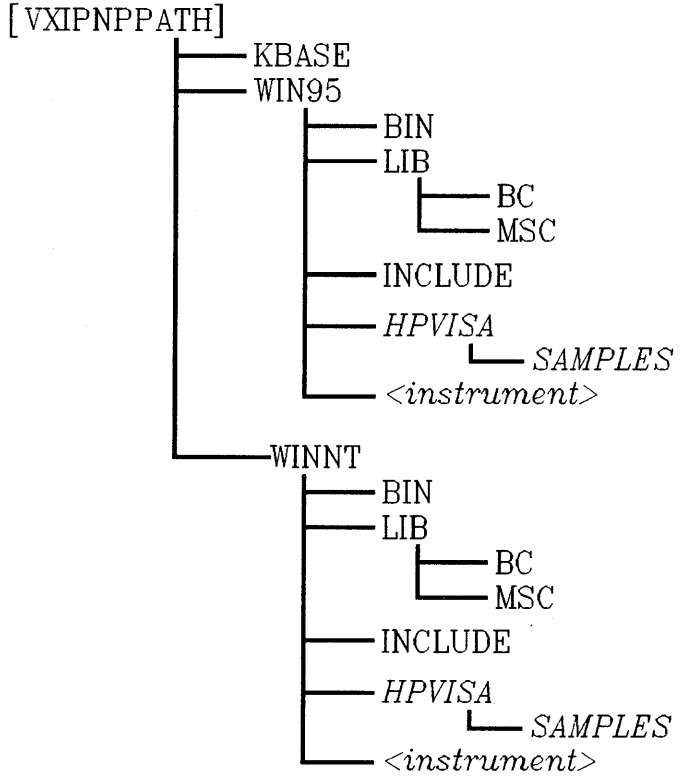
HP VISA System Information

HP VISA System Information

This appendix provides information about the VISA software files. This information can be used as reference, or for removing the VISA software from your system, if necessary.

Windows Directory Structure

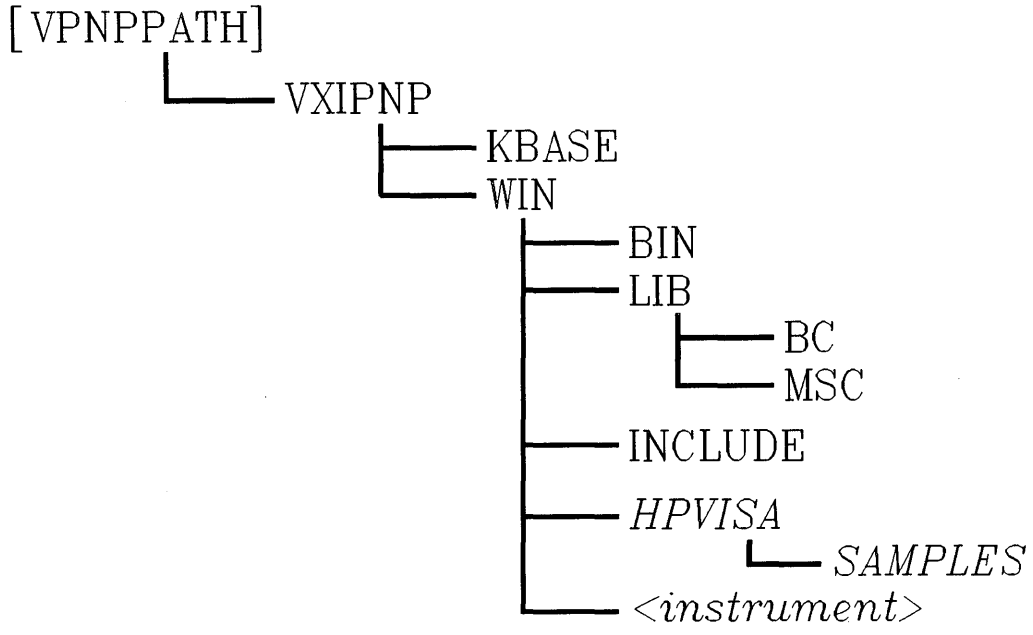
The VXI*plug&play* alliance defines directory structures to be used with the Windows system framework. As shown the following directory structure, 32-bit VISA is automatically installed into either the **WIN95** subdirectory on Windows 95, or the **WINNT** subdirectory on Windows NT. The [*VXIPNPPATH*] is an optional path that you can change during the software installation.



Windows Directory Structure for 32-bit VISA

The `VISA32.DLL` and `HPVISA32.DLL` files are stored in the `\WINDOWS\SYSTEM` subdirectory.

As shown in the following directory structure, 16-bit VISA on Windows 95 is automatically installed into the `WIN` subdirectory. The [*VPNPPATH*] is an optional path that you can change during the software installation.

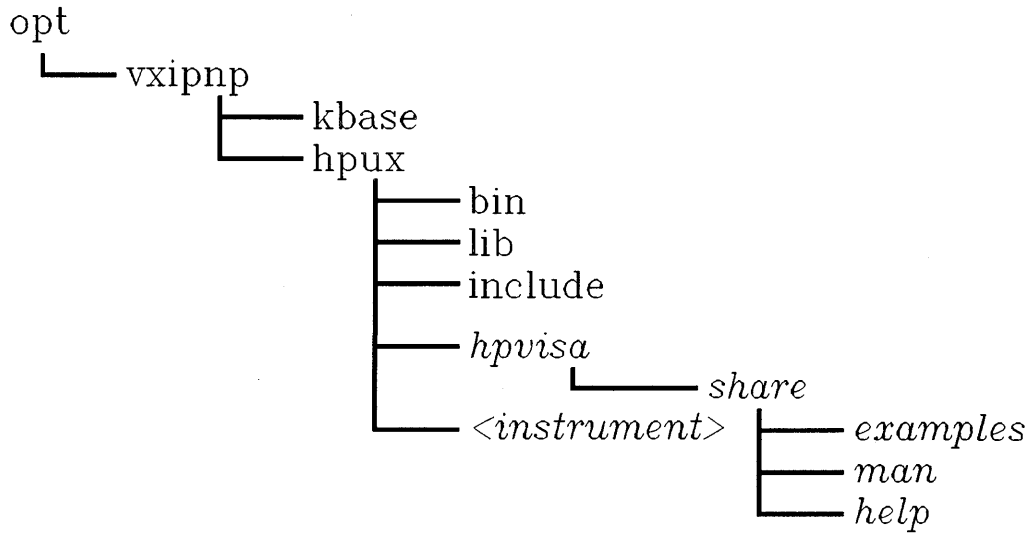


Windows Directory Structure for 16-bit VISA

The `VISA.DLL` file is stored in the `\WINDOWS\SYSTEM` subdirectory.

UNIX Directory Structure

The VXI*plug&play* alliance defines a directory structure to be used with the UNIX system framework. VISA is automatically installed into the following directory structure on HP-UX 10.20. The [*opt*] is an optional path that you can change during the software installation.



UNIX Directory Structure

About the Directories

The HPVISA Subdirectory

Any VISA README files, help files, and HP specific DLLs can be found in the `HPVISA` subdirectory.

Include Files

The `VISA.H`, `VISATYPE.H`, and `VPPTYPE.H` include files can be found in the `INCLUDE` subdirectory.

Libraries

A VISA library is provided for Microsoft and Borland compilers on Windows, and the C compiler for HP-UX. You must use the library for your system.

Sample Programs

Sample programs are provided for the Windows or UNIX operating system, depending on which you have installed. The VISA sample programs can be found in the **HPVISA\SAMPLES** subdirectory on Windows, or in the **hpvisa/share/examples** subdirectory on HP-UX 10.20.

VXIplug&play Instrument Drivers

All instrument drivers that comply with the VXI*plug&play* specification can be found in the *<instrument>* subdirectory, where *<instrument>* is the base directory of the instrument driver.

B

HP VISA Attributes

HP VISA Attributes

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.

Attributes are also local or global. A local attribute only affects the session specified. A global attribute affects the specified device from any session.

For descriptions of all the attributes and how to use them, see the "Using Attributes" section of Chapter 4, "Programming with HP VISA".

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_MAX_QUEUE_LENGTH	RW*	Local	ViUInt32	1h to 32,767 (50 default)
VI_ATTR_RM_SESSION	RO	Local	ViSession	N/A
VI_ATTR_RSRC_IMPL_VERSION	RO	Global	ViVersion	0h to FFFFFFFFh
VI_ATTR_RSRC_LOCK_STATE	RO	Global	ViAccessMode	VI_NO_LOCK (default) VI_EXCLUSIVE_LOCK VI_SHARED_LOCK
VI_ATTR_RSRC_MANF_ID	RO	Global	ViUInt16	0h to 3FFFh
VI_ATTR_RSRC_MANF_NAME	RO	Global	ViString	N/A
VI_ATTR_RSRC_NAME	RO	Global	ViRsrc	N/A
VI_ATTR_RSRC_SPEC_VERSION	RO	Global	ViVersion	00100000h (default)
VI_ATTR_USER_DATA	RW	Local	ViAddr	N/A

* For VISA 1.0, this attribute becomes RO (read only) once **viEnableEvent** has been called for the first time.

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_INTF_NUM	RO	Global	ViUInt16	0 to FFFFh (0 default)
VI_ATTR_INTF_TYPE	RO	Global	ViUInt16	VI_INTF_VXI VI_INTF_GPIB VI_INTF_GPIB_VXI VI_INTF_ASRL
VI_ATTR_IO_PROT	RW	Local	ViUInt16	VI_NORMAL (default) VI_FDC VI_HS488
VI_ATTR_RD_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_DISABLE (default)
VI_ATTR_SEND_END_EN	RW	Local	ViBoolean	VI_TRUE (default) VI_FALSE
VI_ATTR_SUPPRESS_END_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE (default)
VI_ATTR_TERMCHAR	RW	Local	ViUInt8	0 to FFh (0Ah default)
VI_ATTR_TERMCHAR_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE (default)
VI_ATTR_TMO_VALUE	RW	Local	ViUInt32	VI_TMO_IMMEDIATE 1 to FFFFFFFEh VI_TMO_INFINITE (2000 milliseconds default)
VI_ATTR_TRIG_ID	RW*	Local	ViInt16	VI_TRIG_SW (default) VI_TRIG_TTL0 to VI_TRIG_TTL7 VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_ATTR_WR_BUF_OPER_MODE	RW	Local	ViUInt16	VI_FLUSH_ON_ACCESS VI_FLUSH_WHEN_FULL (default)

* The attribute VI_ATTR_TRIG_ID is RW (readable and writable) when the corresponding session is *not* enabled to receive trigger events. When the session is enabled to receive trigger events, this attribute is RO (read only).

HP VISA Interface Specific Instrument Attributes

GPIB and GPIB-VXI Interfaces

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_GPIB_PRIMARY_ADDR	RO	Global	ViUInt16	0 to 30
VI_ATTR_GPIB_SECONDARY_ADDR	RO	Global	ViUInt16	0 to 30 VI_NO_SEC_ADDR

VXI and GPIB-VXI Interfaces

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_CMDR_LA	RO	Global	ViInt16	0 to 255
VI_ATTR_DEST_INCREMENT	RW	Local	ViInt32	0 to 1 (1 default)
VI_ATTR_FDC_CHNL	RW	Local	ViUInt16	0 to 7
VI_ATTR_FDC_GEN_SIGNAL_EN	RW	Local	ViBoolean	VI_TRUE VI_FALSE (default)
VI_ATTR_FDC_MODE	RW	Local	ViUInt16	VI_FDC_NORMAL (default) VI_FDC_STREAM
VI_ATTR_FDC_USE_PAIR	RW	Local	ViBoolean	VI_TRUE VI_FALSE (default)
VI_ATTR_IMMEDIATE_SERV	RO	Global	ViBoolean	VI_TRUE VI_FALSE
VI_ATTR_MAINFRAME_LA	RO	Global	ViInt16	0 to 255 VI_UNKNOWN_LA
VI_ATTR_MANF_ID	RO	Global	ViUInt16	0 to FFFh
VI_ATTR_MEM_BASE	RO	Global	ViBusAddress	N/A
VI_ATTR_MEM_SIZE	RO	Global	ViBusSize	N/A
VI_ATTR_MEM_SPACE	RO	Global	ViUInt16	VI_A16_SPACE (default) VI_A24_SPACE VI_A32_SPACE
VI_ATTR_MODEL_CODE	RO	Global	ViUInt16	0 to FFFFh
VI_ATTR_SLOT	RO	Global	ViInt16	0 to 12 VI_UNKNOWN_SLOT
VI_ATTR_SRC_INCREMENT	RW	Local	ViInt32	0 to 1 (1 default)

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_VXI_LA	RO	Global	ViInt16	0 to 255
VI_ATTR_WIN_ACCESS	RO	Local	ViUInt16	VI_NMAPPED VI_USE_OPERS VI_DEREF_ADDR
VI_ATTR_WIN_BASE_ADDR	RO	Local	ViBusAddress	N/A
VI_ATTR_WIN_SIZE	RO	Local	ViBusSize	N/A

GPIB-VXI Interface

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_INTF_PARENT_NUM	RO	Global	ViUInt16	0 to FFFFh

ASRL Interface

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_ASRL_AVAIL_NUM	RO	Global	ViUInt32	0 to FFFFFFFFh
VI_ATTR_ASRL_BAUD	RW	Global	ViUInt32	0 to FFFFFFFFh (9600 default)
VI_ATTR_ASRL_DATA_BITS	RW	Global	ViUInt16	5 to 8 (8 default)
VI_ATTR_ASRL_END_IN	RW	Local	ViUInt16	VI_ASRL_END_NONE VI_ASRL_END_LAST_BIT VI_ASRL_END_TERMCHAR (default)
VI_ATTR_ASRL_END_OUT	RW	Local	ViUInt16	VI_ASRL_END_NONE (default) VI_ASRL_END_LAST_BIT VI_ASRL_END_BREAK
VI_ATTR_ASRL_FLOW_CNTRL	RW	Global	ViUInt16	VI_ASRL_FLOW_NONE (default) VI_ASRL_FLOW_XON_XOFF VI_ASRL_FLOW_RTS_CTS
VI_ATTR_ASRL_PARITY	RW	Global	ViUInt16	VI_ASRL_PAR_NONE (default) VI_ASRL_PAR_ODD VI_ASRL_PAR_EVEN VI_ASRL_PAR_MARK VI_ASRL_PAR_SPACE
VI_ATTR_ASRL_STOP_BITS	RW	Global	ViUInt16	VI_ASRL_STOP_ONE (default) VI_ASRL_STOP_TWO

Attribute Name	RO or RW	Local or Global	Data Type	Range
VI_ATTR_BUFFER	RO	Local	ViBuf	N/A
VI_ATTR_EVENT_TYPE	RO	Local	ViEventType	VI_EVENT_SERVICE_REQ VI_EVENT_VXI_SIGP VI_EVENT_TRIG VI_EVENT_IO_COMPLETION
VI_ATTR_JOB_ID	RO	Local	ViJobId	N/A
VI_ATTR_RECV_TRIG_ID	RO	Local	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7 VI_TRIG_ECL0 to VI_TRIG_ECL1
VI_ATTR_RET_COUNT	RO	Local	ViUInt32	0 to FFFFFFFFh
VI_ATTR_SIGP_STATUS_ID	RO	Local	ViUInt16	0 to FFFFh
VI_ATTR_STATUS	RO	Local	ViStatus	N/A

NOTE

The VI_EVENT_VXI_SIGP and VI_EVENT_TRIG events are not supported with the GPIB-VXI interface.

C

HP VISA Completion and Error Codes

HP VISA Completion and Error Codes

This appendix lists the VISA completion and error codes. The codes are presented in two different ways. The completion and error codes are listed:

- In alphabetical order for easy look up.
- According to the VISA function that returns the codes. You can use this list to determine what type of codes to expect from each VISA function.

The following tables list the completion and error codes for VISA in alphabetical order for easy look up.

VISA Completion Codes and Descriptions

Completion Code	Description
VI_SUCCESS	Operation completed successfully.
VI_SUCCESS_EVENT_DIS	The specified event is already disabled.
VI_SUCCESS_EVENT_EN	The specified event is already enabled for at least one of the specified mechanisms.
VI_SUCCESS_MAX_CNT	The number of bytes specified were read.
VI_SUCCESS_NESTED_EXCLUSIVE	The specified access mode was successfully acquired, and this session has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The specified access mode was successfully acquired, and this session has nested shared locks.
VI_SUCCESS_QUEUE_EMPTY	The event queue was empty while trying to discard queued events.
VI_SUCCESS_QUEUE_NEMPTY	The event queue is not empty.
VI_SUCCESS_SYNC	The read or write operation performed synchronously.
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_WARN_NSUP_ATTR_STATE	The attribute state is not supported by this resource.
VI_WARN_NSUP_BUF	The specified buffer is not supported.
VI_WARN_UNKNOWN_STATUS	The status code passed to the function was unable to be interpreted.

VISA Error Codes and Descriptions

Error Code	Description
VI_ERROR_ALLOC	Insufficient system resources to open a session or to allocate the buffer(s) or memory block of the specified size.
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_ATTR_READONLY	The attribute specified is read-only.
VI_ERROR_BERR	A bus error occurred during transfer.
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures for this session.
VI_ERROR_HNDLR_NINSTALLED	A handler is not currently installed for the specified event. The session cannot be enabled for the VI_HNDLR mode of the callback mechanism.
VI_ERROR_INP_PROT_VIOL	Input protocol error occurred during transfer.
VI_ERROR_INV_ACCESS_KEY	The *requestedKey* value passed in is not a valid access key to the specified resource.
VI_ERROR_INV_ACC_MODE	The access mode specified is invalid.
VI_ERROR_INV_CONTEXT	The event context specified is invalid.
VI_ERROR_INV_DEGREE	The specified degree is invalid.
VI_ERROR_INV_EVENT	The event type specified is invalid for the specified resource.
VI_ERROR_INV_EXPR	The expression specified is invalid.
VI_ERROR_INV_FMT	The format specifier is invalid for the current argument.
VI_ERROR_INV_HNDLR_REF	The specified handler reference and/or the user context value does not match the installed handler.
VI_ERROR_INV_JOB_ID	The specified job identifier is invalid.
VI_ERROR_INV_LENGTH	The length specified is invalid.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_MASK	The system cannot set the buffer for the given mask, or the specified mask does not specify a valid flush operation on the read/write resource.
VI_ERROR_INV_MECH	The mechanism specified for the event is invalid.

VISA Error Codes and Descriptions (continued)

Error Code	Description
VI_ERROR_INV_OBJECT	The object reference is invalid.
VI_ERROR_INV_OFFSET	The offset specified is invalid.
VI_ERROR_INV_PROT	The protocol specified is invalid.
VI_ERROR_INV_RSRC_NAME	The resources specified are invalid.
VI_ERROR_INV_SESSION	The session specified is invalid.
VI_ERROR_INV_SETUP	The setup specified is invalid, possibly due to attributes being set to an inconsistent state, or some implementation-specific configuration file is corrupt or does not exist.
VI_ERROR_INV_SIZE	The specified size is invalid.
VI_ERROR_INV_SPACE	The address space specified is invalid.
VI_ERROR_IO	Could not perform read/write function because of an I/O error, or an unknown I/O error occurred during transfer.
VI_ERROR_LINE_IN_USE	The specified trigger line is in use.
VI_ERROR_MEM_NSHARED	The device does not export any memory.
VI_ERROR_NCIC	The session is referring to something other than the controller in charge.
VI_ERROR_NIMPL_OPER	The given operation is not implemented.
VI_ERROR_NLISTENERS	No listeners are detected. (Both NRFD and NDAC are deasserted.)
VI_ERROR_NSUP_ATTR	The attribute specified is not supported by the specified resource.
VI_ERROR_NSUP_ATTR_STATE	The state specified for the attribute is not supported.
VI_ERROR_NSUP_FMT	The format specifier is not supported for the current argument type.
VI_ERROR_NSUP_OFFSET	The offset specified is not accessible.
VI_ERROR_NSUP_OPER	The operation specified is not supported in the given session.
VI_ERROR_NSUP_WIDTH	The specified width is not supported by this hardware.
VI_ERROR_QUEUE_ERROR	Unable to queue read or write operation.
VI_ERROR_OUTP_PROT_VIOL	Output protocol error occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	A violation of raw read protocol occurred during a transfer.
VI_ERROR_RAW_WR_PROT_VIOL	A violation of raw write protocol occurred during a transfer.

VISA Error Codes and Descriptions (continued)

Error Code	Description
VI_ERROR_RSRC_LOCKED	The specified operation could not be performed because the resource identifed by *vi* has been locked for this kind of access.
VI_ERROR_RSRC_NFOUND	The expression specified does not match any device, or resource was not found.
VI_ERROR_SRQ_NOCCURED	A service request has not been received for the session.
VI_ERROR_SYSTEM_ERROR	Unknown system error.
VI_ERROR_TMO	The operation failed to complete within the specified timeout period.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.

Completion and Error Codes for Each HP VISA Function

The following lists the VISA functions in alphabetical order, with the associated completion and error codes for each function.

viAssertTrigger(*vi,protocol*)

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

viClear(*vi*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).

viClose(*vi*)

Codes	Description
VI_SUCCESS	Session closed successfully.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.

viDisableEvent(*vi,eventType,mechanism*)

Codes	Description
VI_SUCCESS	Event disabled successfully.
VI_SUCCESS_EVENT_DIS	Specified event is already disabled for at least one of the specified mechanisms.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

viDiscardEvents(*vi,eventType,mechanism*)

Codes	Description
VI_SUCCESS	Event queue flushed successfully.
VI_SUCCESS_QUEUE_EMPTY	Operation completed successfully, but queue empty.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.

viEnableEvent(*vi,eventType,mechanism,context*)

Codes	Description
VI_SUCCESS	Event enabled successfully.
VI_SUCCESS_EVENT_EN	The specified event is already enabled for at least one of the specified mechanisms.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	The specified event type is not supported by the resource.
VI_ERROR_INV_MECH	Invalid mechanism specified.
VI_ERROR_INV_CONTEXT	Invalid event context specified.
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.

viFindNext(*findList,instrDesc*)

Codes	Description
VI_SUCCESS	Resource(s) found.
VI_ERROR_INV_SESSION	The given *findList* is not a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *findList* does not support this function.
VI_ERROR_RSRC_NFOUND	There are no more matches.

viFindRsrc(*sesn,expr,findList,retcnt,instrDesc*)

Codes	Description
VI_SUCCESS	Resource(s) found.
VI_ERROR_INV_SESSION	The given *sesn* is not a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *sesn* does not support this function.
VI_ERROR_INV_EXPR	Invalid expression specified for search.
VI_ERROR_RSRC_NFOUND	Specified expression does not match any devices.

viFlush(*vi,mask*)

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

`viGetAttribute`(*vi,attribute,attrState*)

Codes	Description
VI_SUCCESS	Resource attribute retrieved successfully.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced resource.

`viIn8`(*vi,space,offset,val8*)
`viIn16`(*vi,space,offset,val16*)
`viIn32`(*vi,space,offset,val32*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.

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

Codes	Description
VI_SUCCESS	Event handler installed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not defined by the resource.
VI_ERROR_INV_HNDLR_REF	The given handler reference is invalid.
VI_ERROR_HNDLR_NINSTALLED	The handler was not installed. This may be returned if an application attempts to install multiple handlers for the same event on the same session.

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

Codes	Description
VI_SUCCESS	The specified access mode was successfully acquired.
VI_SUCCESS_NESTED_EXCLUSIVE	The specified access mode was successfully acquired, and this session has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The specifed access mode was successfully acquired, and this session has nested shared locks.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	The specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested.
VI_ERROR_INV_LOCK_TYPE	The specified type of lock is not supported by this resource.
VI_ERROR_INV_ACCESS_KEY	The *requestedKey* value passed is not a valid access key to the specified resource.
VI_ERROR_TMO	The specified type of lock could not be obtained within the specified timeout period.

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

Codes	Description
VI_SUCCESS	Map successful.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified region is not accessible from this hardware.
VI_ERROR_TMO	Could not acquire resource or perform mapping before the timer expired.
VI_ERROR_INV_SIZE	Invalid size of window specified.
VI_ERROR_ALLOC	Unable to allocate window of at least the requested size.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.

`viMemAlloc`(*vi, size, offset*)

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

`viMemFree`(*vi,offset*)

Codes	Description
VI_SUCCESS	The operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this operation.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_WINDOW_MAPPED	The specified offset is currently in use by `viMapAddress`.

`viMoveIn8`(*vi,space,offset,length,buf8*)
`viMoveIn16`(*vi,space,offset,length,buf16*)
`viMoveIn32`(*vi,space,offset,length,buf32*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_LENGTH	Invalid length specified.

`viMoveOut8`(*vi,space,offset,length,buf8*)
`viMoveOut16`(*vi,space,offset,length,buf16*)
`viMoveOut32`(*vi,space,offset,length,buf32*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
VI_ERROR_INV_LENGTH	Invalid length specified.

`viOpen`(*sesn,rsrcName,accessMode,timeout,vi*)

Codes	Description
VI_SUCCESS	Session opened successfully.
VI_ERROR_INV_SESSION	The given *sesn* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *sesn* does not support this function. For VISA, this operation is supported only by the Default Resource Manager session.
VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
VI_ERROR_INV_ACC_MODE	Invalid access mode.
VI_ERROR_RSRC_NFOUND	Insufficient location information or resource not present in the system.
VI_ERROR_ALLOC	Insufficient system resources to open a session.

`viOpenDefaultRM(`*sesn*`)`

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

`viOut8(`*vi,space,offset,val8*`)`
`viOut16(`*vi,space,offset,val16*`)`
`viOut32(`*vi,space,offset,val32*`)`

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SPACE	Invalid address space specified.
VI_ERROR_INV_OFFSET	Invalid offset specified.
VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.

`viPeek8(`*vi,addr,val8*`)`
`viPeek16(`*vi,addr,val16*`)`
`viPeek32(`*vi,addr,val32*`)`
These functions do not return any completion or error codes.

`viPoke8(`*vi,addr,val8*`)`
`viPoke16(`*vi,addr,val16*`)`
`viPoke32(`*vi,addr,val32*`)`
These functions do not return any completion or error codes.

`viPrintf`(*vi,writeFmt,arg1,arg2*)

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

`viQueryf`(*vi,writeFmt,readFmt,arg1,arg2*)

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

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

Codes	Description
VI_SUCCESS	The operation completed successfully and the END indicator was received (for interfaces that have END indicators).
VI_SUCCESS_TERM_CHAR	The specified termination character was read.
VI_SUCCESS_MAX_CNT	The number of bytes read is equal to *count*.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_ASRL_PARITY	A parity error occurred during transfer.
VI_ERROR_ASRL_FRAMING	A framing error occurred during transfer.
VI_ERROR_ASRL_OVERRUN	An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived.
VI_ERROR_IO	An unknown I/O error occurred during transfer.

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

Codes	Description
VI_SUCCESS	Asynchronous read operation successfully queued.
VI_SUCCESS_SYNC	Read operation performed synchronously.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_QUEUE_ERROR	Unable to queue read operation.

`viReadSTB`(*vi,status*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_SRQ_NOCCURRED	Service request has not been received for the session.
VI_ERROR_TMO	Timeout expired before function completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).

`viScanf`(*vi,readFmt,arg1,arg2*)

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

`viSetAttribute`(*vi,attribute,attrState*)

Codes	Description
VI_SUCCESS	All attribute values set successfully.
VI_WARN_NSUP_ATTR_STATE	Although the specified state of the attribute is valid, it is not supported by this resource implementation
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	The specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_NSUP_ATTR	The specified attribute is not defined by the referenced resource.
VI_ERROR_NSUP_ATTR_STATE	The specified state of the attribute is not valid, or is not supported as defined by the resource.
VI_ERROR_ATTR_READONLY	The specified attribute is read-only.

`viSetBuf(`*vi,mask,size*`)`

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

`viStatusDesc(`*vi,status,desc*`)`

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

`viTerminate(`*vi,degree,jobId*`)`

Codes	Description
VI_SUCCESS	Request serviced successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_DEGREE	Invalid degree specified.
VI_ERROR_INV_JOB_ID	Invalid job identifier specified.

viUninstallHandler(*vi,eventType,handler,userHandle*)

Codes	Description
VI_SUCCESS	Event handler successfully uninstalled.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_INV_HNDLR_REF	Either the specified handler reference or the user context value (or both) does not match any installed handler.

viUnlock(*vi*)

Codes	Description
VI_SUCCESS	The lock was successfully relinquished.
VI_SUCCESS_NESTED_EXCLUSIVE	The call succeeded, but this session still has nested exclusive locks.
VI_SUCCESS_NESTED_SHARED	The call succeeded, but this session still has nested shared locks.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_SESN_NLOCKED	The current session did not have any lock on the resource.

viUnmapAddress(*vi*)

Codes	Description
VI_SUCCESS	Operation completed successfully.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.

`viVPrintf`(*vi,writeFmt,params*)

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

`viVQueryf`(*vi,writeFmt,readFmt,params*)

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

`viVScanf`(*vi,readFmt,params*)

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

`viWaitOnEvent`(*vi,ineventType,timeout,outEventType,outcontext*)

VI_SUCCESS	Wait terminated successfully on receipt of an event occurrence. The queue is empty.
VI_SUCCESS_QUEUE_NEMPTY	Wait terminated successfully on receipt of an event notification. There is still at least one more event occurrence available for this session.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
VI_ERROR_TMO	Specified event did not occur within the specified time period.

viWrite(*vi,buf,count,retCount*)

Codes	Description
VI_SUCCESS	Transfer completed.
VI_ERROR_INV_SESSION	The given *vi* does not identify a valid session.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_NSUP_OPER	The given *vi* does not support this function.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_TMO	Timeout expired before operation completed.
VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error occurred during transfer.
VI_ERROR_BERR	Bus error occurred during transfer.
VI_ERROR_INV_SETUP	Unable to start read operation because setup is invalid (due to attributes being set to an inconsistent state).
VI_ERROR_NCIC	*vi* does not refer to an interface that is currently the controller in charge.
VI_ERROR_NLISTENERS	No listeners condition is detected (both NRFD and NDAC are deasserted).
VI_ERROR_IO	An unknown I/O error occurred during transfer.

viWriteAsync(*vi,buf,count,jobId*)

Codes	Description
VI_SUCCESS	Asynchronous write operation successfully queued.
VI_SUCCESS_SYNC	Write operation performed synchronously.
VI_ERROR_INV_SESSION	The given session is invalid.
VI_ERROR_INV_OBJECT	The given object reference is invalid.
VI_ERROR_RSRC_LOCKED	Specified operation could not be performed because the resource identified by *vi* has been locked for this kind of access.
VI_ERROR_QUEUE_ERROR	Unable to queue write operation.

D

HP VISA Type Definitions

HP VISA Type Definitions

This appendix lists the VISA data types and their definitions.

VISA Type Definitions

VISA Data Type	Type Definition	Description
`ViUInt32`	`unsigned long`	A 32-bit unsigned integer.
`ViPUInt32`	`ViUInt32 *`	The location of a 32-bit unsigned integer.
`ViAUInt32`	`ViUInt32 *`	The location of a 32-bit unsigned integer.
`ViInt32`	`signed long`	A 32-bit signed integer.
`ViPInt32`	`ViInt32 *`	The location of a 32-bit signed integer.
`ViAInt32`	`ViInt32 *`	The location of 32-bit signed integer.
`ViUInt16`	`unsigned short`	A 16-bit unsigned integer.
`ViPUInt16`	`ViUInt16 *`	The location of a 16-bit unsigned integer.
`ViAUInt16`	`ViUInt16 *`	The location of a 16-bit unsigned integer.
`ViInt16`	`signed short`	A 16-bit signed integer.
`ViPInt16`	`ViInt16 *`	The location of a 16-bit signed integer.
`ViAInt16`	`ViInt16 *`	The location of 16-bit signed integer.
`ViUInt8`	`unsigned char`	An 8-bit unsigned integer.
`ViPUInt8`	`ViUInt8 *`	The location of an 8-bit unsigned integer.
`ViAUInt8`	`ViUInt8 *`	The location of an 8-bit unsigned integer.
`ViInt8`	`signed char`	An 8-bit signed integer.
`ViPInt8`	`ViInt8 *`	The location of an 8-bit signed integer.
`ViAInt8`	`ViInt8 *`	The location of an 8-bit signed integer.
`ViAddr`	`void *`	A type that references another data type.
`ViPAddr`	`ViAddr *`	The location of a `ViAddr`
`ViChar`	`char`	An 8-bit integer representing an ASCII character.
`ViPChar`	`ViChar *`	The location of a `ViChar`.
`ViByte`	`unsigned char`	An 8-bit unsigned integer representing an extended ASCII character.
`ViPByte`	`ViByte *`	The location of a `ViByte`.

VISA Type Definitions (continued)

VISA Data Type	Type Definition	Description
ViBoolean	ViUInt16	A type that is either VI_TRUE or VI_FALSE.
ViPBoolean	ViBoolean *	The location of a ViBoolean.
ViBuf	ViPByte	The location of a block of data.
ViPBuf	ViPByte	The location of a block of data.
ViString	ViPChar	The location of a NULL-terminated ASCII string.
ViPString	ViPChar	The location of a NULL-terminated ASCII string.
ViStatus	ViInt32	Values that correspond to VISA-defined completion and error codes.
ViPStatus	ViStatus *	The location of the completion and error codes.
ViRsrc	ViString	A ViString type.
ViPRsrc	ViString	A ViString type.
ViAccessMode	ViUInt32	Specifies the different mechanisms that control access to a resource.
ViBusAddress	ViUInt32	Represents the system dependent physical address.
ViBusSize	ViUInt32	Represents the system dependent physical address size.
ViAttr	ViUInt32	Identifies an attribute.
ViVersion	ViUInt32	Specifies the current version of the resource.
ViPVersion	ViVersion *	The location of ViVersion.
ViAttrState	ViUInt32	Specifies the type of attribute.
ViPAttrState	void *	The location of ViAttrState.
ViVAList	va_list	The location of a list of variable number of parameters of differing types.
ViEventType	ViUInt32	Specifies the type of event.
ViPEventType	ViEventType *	The location of a ViEventType.
ViEventFilter	ViUInt32	Specifies filtering masks or other information unique to an event.

VISA Type Definitions (continued)

VISA Data Type	Type Definition	Description
`ViObject`	`ViUInt32`	Contains attributes and can be closed when no longer needed.
`ViPObject`	`ViObject *`	The location of a `ViObject`.
`ViSession`	`ViObject`	Specifies the information necessary to manage a communication channel with a resource.
`ViPSession`	`ViSession *`	The location of a `ViSession`.
`ViFindList`	`ViObject`	Contains a reference to all resources found during a search operation.
`ViPFindList`	`ViFindList *`	The location of a `ViFindList`.
`ViEvent`	`ViObject`	Contains information necessary to process an event.
`ViPEvent`	`ViEvent *`	The location of a `ViEvent`.
`ViHndlr`	`ViStatus (*) (ViSession, ViEventType, ViEvent, ViAddr)`	A value representing an entry point to an operation for use as a callback.
`ViReal32`	`float`	A 32-bit, single-precision value.
`ViPReal32`	`ViReal32 *`	The location of a 32-bit, single-precision value.
`ViReal64`	`double`	A 64-bit, double-precision value.
`ViPReal64`	`ViReal64 *`	The location of a 64-bit, double-precision value.
`ViJobId`	`ViUInt32`	The location of a variable that will be set to the job identifier.
`ViKeyId`	`ViPString`	The location of a string.

E

Editing the HP VISA Configuration

Editing the HP VISA Configuration

When the HP I/O Libraries are configured, certain values are used as defaults in the VISA configuration. In some cases the default values will affect your system performance. If you are having system performance problems, you may need to edit the configuration and change some default values. This appendix describes how to edit the configuration for VISA on Windows 95 and Windows NT, and on HP-UX.

On Windows 95 and Windows NT

When you first configured the HP I/O Libraries, the default configuration specified that all VISA devices would be identified at run-time. However, this is not ideal for all users. If you are experiencing performance problems, particularly during `viOpenDefaultRM`, you may want to change the VISA configuration to identify devices during configuration. This may be especially helpful if you are using a VISA LAN client.

To edit the default VISA configuration on Windows 95 or Windows NT, do the following:

1. If you have not already done so, start up Windows 95 or Windows NT.
2. Run the `I/O Config` utility, which is located in the `HP I/O Libraries` program group.
3. Select the interface you wish to configure from the `Configured Interfaces` box, and click on the (Edit) button.

   The Interface Edit window is now displayed.
4. Click on the (Edit VISA Config) button at the bottom of the window.

   The dialog box which allows you to add devices is now displayed.
5. You can now manually identify devices by clicking on the (Add Device) button and entering the device address.

NOTE

If you wish to turn off the default of identifying devices at run-time, you must un-select the `Identify devices at run-time` box at the top of the dialog box.

You may also click on the (Auto Add Devices) button at the bottom of the screen to automatically check for devices at this time. If you select this button, the utility will prompt you to make sure all devices are connected and turned on. Once this process is complete, you may edit this list with the (Add Device) and (Remove Device) buttons.

6. Once you have completed adding or removing devices, select the (OK) button to exit the window. Then exit the `I/O Config` utility to save the changes you have made.

On HP-UX

When you first configured the HP I/O Libraries, the default configuration specified that all VISA devices would be identified at run-time. However, this is not ideal for all users. If you are experiencing performance problems, particularly during **viOpenDefaultRM**, you may want to change the VISA configuration to identify devices during configuration.

To edit the default VISA configuration on HP-UX, use the following command to run the **visacfg** utility:

```
/opt/vxipnp/hpux/hpvisa/visacfg
```

Follow the instructions provided in the utility. When prompted, select the (Add Device) button and add all devices that will be used.

Glossary

Glossary

address
A string uniquely identifying a particular device on an interface.

attributes
Values that determine the state of a resource. The operational state of some attributes can be changed.

bus error
An action that occurs when access to a given address fails either because no register exists at the given address, or the register at the address refuses to respond.

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

device
A unit that receives commands from a controller. Typically a device is an instrument but could also be a computer acting in a non-controller role, or another peripheral such as a printer or plotter.

device driver
A segment of software code that communicates with a device. It may either communicate directly with a device by reading to and writing from registers, or it may communicate through an interface driver.

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

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

instrument
A device that accepts commands and performs a test or measurement function.

interface
A connection and communication media between devices and controllers, including mechanical, electrical, and protocol connections.

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

mapping
An operation that returns a pointer to a specified section of an address space and makes the specified range of addresses accessible to the requester.

process
An operating system object containing one or more threads of execution that share a data space. A multi-process system is a computer system that allows multiple programs to execute simultaneously, each in a separate process environment. A single-process system is a computer system that allows only a single program to execute at a given point in time.

register
An address location that controls or monitors hardware.

resource
An instrument while using VISA.

session
An instance of a communications path between a software element and a resource.

SRQ
Service Request. An asynchronous request (an interrupt) from a remote device indicating that the device requires servicing.

status byte
A byte of information returned from a remote device showing the current state and status of the device.

thread

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

VISA

Virtual Instrument Software Architecture. VISA is a common I/O library where software from different vendors can run together on the same platform.

Index

A

Addressing
  devices, 4-7
  over LAN, 6-10
  sessions, 4-7
Applications, building, 2-6
Argument length modifier, 4-17
, Array size, 4-18
ASRL, attributes, 4-31, B-8
Attributes
  ASRL, 4-31, B-8
  changing, 4-26
  events, 4-32, 4-35, B-9
  generic INSTR, 4-28, B-4
  GPIB, 4-29, B-5
  GPIB-VXI, 4-29, 4-31, B-5, B-6, B-7
  interface specific, 4-29, B-5
  reading, 4-26
  reading for events, 4-36
  resource, 4-27, B-3
  serial, 4-31, B-8
  setting VXI trigger lines, 5-25
  VXI, 4-29, 5-23, B-6

B

Buffers
  flushing, 4-22
  formatted I/O, 4-22
Building DLLs, 2-6

C

Callbacks and events, 4-33, 4-37
Closing sessions, 4-9
Compiling
  16-bit, 2-8
  32-bit, 2-7
  in HP-UX, 3-6
Completion codes, C-3
Configuration
  editing VISA, E-2
  LAN, 6-9
Conversion characters, 4-19
Conversion of formatted I/O, 4-15

D

Declarations file, 4-3
Default resource manager, 4-4
Device sessions
  addressing, 4-7
  closing, 4-9
  opening, 4-5
Directory structure
  HP-UX, A-5
  Windows, A-3
DLLs, building , 2-6
Documentation, 1-7

E

Editing VISA configuration, E-2
Enable events
  for callback, 4-39
  for queuing, 4-45

Error codes, C-3
Error messages, logging
  on HP-UX, 3-7
  on Windows 95, 2-11
  on Windows NT, 2-11
Error trapping
  instrument errors, 4-50
  VISA errors, 4-49
Event attributes, 4-32, 4-35, B-9
Event handler, 4-38
Events
  attributes, 4-35
  callback, 4-33, 4-37
  enable for callback, 4-39
  enable for queuing, 4-45
  handlers, 4-33
  hardware triggers, 4-33
  interrupts, 4-33
  queuing, 4-33, 4-45
  reading attributes, 4-36
  SRQs, 4-33
  wait on event, 4-46
Event Types
  VI_EVENT_IO_COMPLETION, 4-35
  VI_EVENT_SERVICE_REQ, 4-35
  VI_EVENT_TRIG, 4-35
  VI_EVENT_VXI_SIGP, 4-35
Event Viewer utility, 2-11
evnthdlr.c example, 4-40
evntqueu.c example, 4-47
Examples
  directory location, A-7
  evnthdlr.c, 4-40
  evntqueu.c, 4-47
  formatio.c, 4-20
  formatio.c over LAN, 6-11
  gpibvxi.c, 5-10
  gpibvxil.c, 5-16
  idn.c, 2-3, 3-3
  lockexcl.c, 4-56
  lockshr.c, 4-58
  nonfmtio.c, 4-24
  running on HP-UX, 3-8
  running on Windows, 2-12
  srqhdlr.c, 4-42
  vxihl.c, 5-9
  vxill.c, 5-14
Exclusive locks, 4-53, 4-55

F

Field width, 4-15
Finding resources, 4-10
Flushing buffers, 4-22
formatio.c example, 4-20
formatio.c example over LAN, 6-11
Format string, 4-22
Formatted I/O
  argument length modifier, 4-17
  , array size, 4-18
  buffers, 4-22
  conversion, 4-15
  conversion characters, 4-19
  description, 4-13
  field width, 4-15
  format string, 4-22
  functions, 4-14
  modifiers, 4-15
  . precision, 4-16
  special characters, 4-18
Functions
  formatted I/O, 4-14
  non-formatted I/O, 4-23
  viAssertTrigger, 7-7
  viClear, 7-9
  viClose, 4-9, 7-11
  viDisableEvent, 4-40, 7-13
  viDiscardEvents, 7-16
  viEnableEvent, 4-39, 4-45, 7-18
  viEventHandler, 7-21
  viFindNext, 4-10, 7-24
  viFindRsrc, 4-10, 7-25
  viFlush, 4-22, 7-27

viGetAttribute, 4-26, 7-30
viIn16, 5-8, 7-32
viIn32, 5-8, 7-32
viIn8, 5-8, 7-32
viInstallHandler, 4-37, 7-34
viLock, 4-51, 7-36
viMapAddress, 5-12, 5-13, 7-41
viMemAlloc, 7-44
viMemFree, 7-46
viMoveIn16, 5-8, 7-47
viMoveIn32, 5-8, 7-47
viMoveIn8, 5-8, 7-47
viMoveOut16, 5-8, 7-50
viMoveOut32, 5-8, 7-50
viMoveOut8, 5-8, 7-50
viOpen, 4-5, 7-53
viOpenDefaultRM, 4-4, 7-55
viOut16, 5-8, 7-57
viOut32, 5-8, 7-57
viOut8, 5-8, 7-57
viPeek16, 5-13, 7-59
viPeek32, 5-13, 7-59
viPeek8, 5-13, 7-59
viPoke16, 5-13, 7-61
viPoke32, 5-13, 7-61
viPoke8, 5-13, 7-61
viPrintf, 4-14, 7-63
viQueryf, 4-14, 7-71
viRead, 4-23, 7-73
viReadAsync, 4-23, 7-76
viReadSTB, 7-78
viScanf, 4-14, 7-80
viSetAttribute, 7-87
viSetBuf, 4-22, 7-89
viStatusDesc, 7-91
viTerminate, 7-92
viUninstallHandler, 7-93
viUnlock, 4-51, 7-95
viUnmapAddress, 5-13, 7-97
viVPrintf, 4-14, 7-98
viVQueryf, 4-14, 7-100
viVScanf, 4-14, 7-102
viWaitOnEvent, 4-46, 7-104
viWrite, 4-23, 7-107
viWriteAsync, 4-23, 7-109

G

Generic INSTR attributes, 4-28, B-4
GPIB
  and SRQs over LAN, 6-18
  attributes, 4-29, B-5
  interface, 5-3
GPIB-VXI
  attributes, 4-29, 4-31, 5-23, B-5, B-6, B-7
  high-level memory functions, 5-5
  interface, 5-3
  low-level memory functions, 5-11
  mapping memory space, 5-12
  message-based devices, 5-4
  programming overview, 5-3
  register-based devices, 5-4
  register programming, 5-6, 5-11
  setting trigger lines, 5-25
  writing to registers, 5-13
gpibvxi.c example, 5-10
gpibvxil.c example, 5-16

H

Handlers, 4-33
  event, 4-38
  installing, 4-37
  prototype, 4-38
Hardware triggers and events, 4-33
Header file, visa.h, 4-3
Help
  HyperHelp on HP-UX, 3-9
  man pages on HP-UX, 3-10
High-level memory functions for VXI, 5-5, 5-6
HP-UX
  compiling , 3-6

directory structure, A-5
linking , 3-6
logging messages, 3-7
online help, 3-9
HPVISA subdirectory, A-6
HyperHelp on HP-UX, 3-9

I

idn.c example, 2-3, 3-3
IEEE Standard, 1-7
Include files, A-6
Installing handlers, 4-37
INSTR, 4-7
Instrument drivers, directory location, A-7
Instrument errors, 4-50
Interfaces
GPIB, 5-3
GPIB-VXI, 5-3
LAN, 6-4
VXI, 5-3
Interface specific attributes, 4-29, B-5
Interrupts and events, 4-33

L

LAN
addressing, 6-10
and SRQs, 6-18
client/server, 6-4
communication, 6-10
configuration, 6-9
networking protocols, 6-7
overview, 6-4
performance, 6-9
servers, 6-8
SICL LAN Protocol, 6-7
signal handling, 6-17
software architecture, 6-6
starting or stopping server, 6-2
TCP/IP Instrument Protocol, 6-7
threads with LAN client, 6-8
timeouts, 6-13
VISA function support, 6-18
LAN client
definition, 6-4
threads used with, 6-8
LAN server
definition, 6-4
description of, 6-8
starting or stopping, 6-2
LAN-to-Instrument Gateway, 6-5
Libraries, 2-6, A-6
Linking
16-bit, 2-8
32-bit, 2-7
in HP-UX, 3-6
Linking to VISA libraries, 2-6
lockexcl.c example, 4-56
Locks
access modes, 4-53
acquiring exclusive lock while holding shared lock, 4-55
examples, 4-56
exclusive, 4-53
lockexcl.c example, 4-56
lockshr.c example, 4-58
nested, 4-56
shared, 4-53, 4-54
types, 4-53
using, 4-51
lockshr.c example, 4-58
Logging messages
on HP-UX, 3-7
on Windows 95, 2-11
on Windows NT, 2-11
Low-level memory functions for VXI, 5-11

M

man pages on HP-UX, 3-10

Memory I/O performance with VXI, 5-17
Memory mapping, 5-12
Memory models, 2-8
Memory space, unmapping, 5-13
Message-based devices, 5-4
`Message Viewer` utility, 2-11
Modifiers, 4-15

N

Nested locks, 4-56
Networking protocols, 6-7
`nonfmtio.c` example, 4-24
Non-formatted I/O
  description, 4-13
  functions, 4-23
  mixing with formatted I/O, 4-23

O

Online help in HP-UX, 3-9
Opening sessions, 4-4
Overview
  VISA, 1-4

P

Performance
  with LAN, 6-9
  with VXI, 5-17
. Precision, 4-16
Protocols, networking, 6-7

Q

Queuing and events, 4-33, 4-45

R

Raw I/O, 4-23
Register-based devices, 5-4
Register programming
  high-level memory functions, 5-6
  low-level memory functions, 5-11
  mapping memory space, 5-12
Resource attributes, 4-27, B-3
Resource manager, 4-4
Resource manager session, 4-4
Resources
  finding, 4-10
  locking, 4-51
Running an example program, 2-12, 3-8

S

Searching for resources, 4-10
Serial, attributes, 4-31, B-8
Servers, LAN, 6-8
Sessions
  addressing, 4-7
  closing, 4-9
  device, 4-5
  LAN, 6-10
  opening, 4-4
  resource manager, 4-4
Shared locks, 4-53, 4-54, 4-55
SICL LAN Networking Protocol, 6-7
Signal handling with LAN, 6-17
Special characters, 4-18
`srqhdlr.c` example, 4-42
SRQs
  and events, 4-33
  over LAN, 6-18
Starting or stopping the LAN Server, 6-2
Starting the resource manager, 4-4

T

TCP/IP Instrument Networking Protocol, 6-7
Threads in 32-bit, 6-8
Timeouts with LAN, 6-13
Trapping errors
  instrument errors, 4-50
  VISA errors, 4-49
Trigger lines, 5-25

Triggers and events, 4-33
Types, VISA, D-2

U

Unmapping memory space, 5-13
Utilities
  Event Viewer, 2-11
  Message Viewer, 2-11

V

viAssertTrigger, 7-7
viClear, 7-9
viClose, 4-9, 7-11
viDisableEvent, 4-40, 7-13
viDiscardEvents, 7-16
viEnableEvent, 4-39, 4-45, 7-18
viEventHandler, 7-21
VI_EVENT_IO_COMPLETION, 4-35
VI_EVENT_SERVICE_REQ, 4-35
VI_EVENT_TRIG, 4-35
VI_EVENT_VXI_SIGP, 4-35
viFindNext, 4-10, 7-24
viFindRsrc, 4-10, 7-25
viFlush, 4-22, 7-27
viGetAttribute, 4-26, 7-30
viIn16, 5-8, 7-32
viIn32, 5-8, 7-32
viIn8, 5-8, 7-32
viInstallHandler, 4-37, 7-34
viLock, 4-51, 7-36
viMapAddress, 5-12, 5-13, 7-41
viMemAlloc, 7-44
viMemFree, 7-46
viMoveIn16, 5-8, 7-47
viMoveIn32, 5-8, 7-47
viMoveIn8, 5-8, 7-47
viMoveOut16, 5-8, 7-50
viMoveOut32, 5-8, 7-50
viMoveOut8, 5-8, 7-50
viOpen, 4-5, 7-53
viOpenDefaultRM, 4-4, 7-55
viOut16, 5-8, 7-57
viOut32, 5-8, 7-57
viOut8, 5-8, 7-57
viPeek16, 5-13, 7-59
viPeek32, 5-13, 7-59
viPeek8, 5-13, 7-59
viPoke16, 5-13, 7-61
viPoke32, 5-13, 7-61
viPoke8, 5-13, 7-61
viPrintf, 4-14, 7-63
viQueryf, 4-14, 7-71
viRead, 4-23, 7-73
viReadAsync, 4-23, 7-76
viReadSTB, 7-78
VISA
  completion codes, C-3
  editing configuration, E-2
  error codes, C-3
  errors, 4-49
  HP-UX support, 1-5
  interfaces on HP-UX, 1-5
  interfaces on Windows, 1-4
  other documentation, 1-7
  overview, 1-4
  programming languages on HP-UX, 1-5
  programming languages on Windows, 1-4
  specification, 1-7
  trigger lines, 5-25
  types, D-2
  users, 1-6
  Windows support, 1-4
visa.h header file, 4-3
viScanf, 4-14, 7-80
viSetAttribute, 7-87
viSetBuf, 4-22, 7-89
viStatusDesc, 7-91
viTerminate, 7-92
viUninstallHandler, 7-93
viUnlock, 4-51, 7-95

viUnmapAddress, 5-13, 7-97
viVPrintf, 4-14, 7-98
viVQueryf, 4-14, 7-100
viVScanf, 4-14, 7-102
viWaitOnEvent, 4-46, 7-104
viWrite, 4-23, 7-107
viWriteAsync, 4-23, 7-109
VXI
  attributes, 4-29, 5-23, B-6
  high-level memory functions, 5-5
  interface, 5-3
  low-level memory functions, 5-11
  mapping memory space, 5-12
  message-based devices, 5-4
  performance, 5-17
  programming overview, 5-3
  register-based devices, 5-4
  register programming, 5-6, 5-11
  setting trigger lines, 5-25
  writing to registers, 5-13
vxihl.c example, 5-9
vxill.c example, 5-14

W

Wait on event, 4-46
Windows
  building applications, 2-6
  building DLLs, 2-6
  directory structure, A-3
  linking to VISA libraries, 2-6
Windows 95
  compiling for 16-bit, 2-8
  compiling for 32-bit, 2-7
  LAN client and threads, 6-8
  linking for 16-bit, 2-8
  linking for 32-bit, 2-7
  logging messages, 2-11
  starting or stopping LAN server, 6-2
  threads in 32-bit, 6-8
Windows NT
  compiling, 2-7
  LAN client and threads, 6-8
  linking, 2-7
  logging messages, 2-11
  starting or stopping LAN server, 6-2
  threads, 6-8
Writing to VXI registers, 5-13

Printed in U.S.A. E0996

E2090-90032