

**Systems Alliance**

**VPP-4.3: The VISA Library**

**June 19, 2015**

**Revision 5.4**



**Systems Alliance**

**VPP-4.3 Revision History**

This section is an overview of the revision history of the VPP-4.3 specification.

**Revision 1.0, December 29, 1995**

Original VISA document. Changes from VISA Transition Library include locking, asynchronous I/O, 32-bit register access, block moves, shared memory operations, and serial interface support.

**Revision 1.1, January 22, 1997**

Added new attributes, error codes, events, and formatted I/O modifiers.

**Revision 2.0, December 5, 1997**

Added error handling event, more formatted I/O operations, more serial attributes and extended searching capabilities.

**Revision 2.0.1, December 4, 1998**

Added new types to visatype.h for instrument drivers. Added new modes to give more robust functionality to viGpibControlREN. Updated information regarding contacting the Alliance.

**Revision 2.2, November 19, 1999**

Added new resource classes for GPIB (INTFC and SERVANT), VXI (BACKPLANE and SERVANT), and TCPIP (INSTR, SOCKET, and SERVANT).

**Revision 3.0 Draft, January 28, 2003**

Added new resource class for USB (INSTR). Added extended parsing capability.

**Revision 3.0, January 15, 2004**

Approved at IVI Board of Directors meeting.

**Revision 4.0 Draft, May 16, 2006**

Added new resource class for PXI (INSTR) to incorporate PXISA extensions. Added 64-bit extensions for register-based operations. Added support for new WIN64 framework.

**Revision 4.0, October 12, 2006**

Approved at IVI Board of Directors meeting.

**Revision 4.1, February 14, 2008**

Updated the introduction to reflect the IVI Foundation organization changes. Replaced Notice with text used by IVI Foundation specifications.

**Revision 4.1, April 14, 2008**

Editorial change to update the IVI Foundation contact information in the Important Information section to remove obsolete address information and refer only to the IVI Foundation web site.

**Revision 4.2, October 16, 2008**

Tightened requirements for resource strings returned by viFindRsrc, viParseRsrc, and viParseRsrcEx to ensure that they return identical strings for use by the new VISA Router component.

**Revision 5.0, June 9, 2010**

Added support for HiSLIP devices under the TCPIP INSTR designation. This includes updates to the resource string and new attributes. Also added format specifiers for the long long type per ANSI C.

**Revision 5.1, October 11, 2012**

Added support extended VXIbus block transfer protocols and trigger capabilities according to VXI-1 4.0. Extensions for PXI INSTR, PXI BACKPLANE.

**Revision 5.4, June 19, 2014**

Added clarifications (rules and observations) to viOpen, viReadAsync, viWriteAsync and viMoveAsync. Added a new error code VI\_ERROR\_LINE\_NRESERVED to facilitate better mapping of PXI-9 trigger error codes. Added clarifications (rules and permissions) to viMapTrigger and viUnmapTrigger. Extended viGpibControlREN to add support for TCPIP devices. Changed the version to 5.4 to ensure that all VISA specifications being voted on at the same time have the same version.

**Revision 5.4 Editorial Change, June 19, 2015**

Added VI\_WARN\_QUEUE\_OVERFLOW as a possible error code returned by viWaitOnEvent. Added clarifications on the effect of the precision modifier with the %f specifier. Added clarifications on the value of failureIndex when viPxiReserveTriggers returns different status codes. Added clarifications on the maximum length of VISA string attributes.

**NOTICE**

VPP-4.3: *The VISA Library* is authored by the IVI Foundation member companies. For a vendor membership roster list, please visit the IVI Foundation web site at www.ivifoundation.org.

The IVI Foundation wants to receive your comments on this specification. You can contact the Foundation through the web site at www.ivifoundation.org.

**Warranty**

The IVI Foundation and its member companies make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The IVI Foundation and its member companies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

**Trademarks**

Product and company names listed are trademarks or trade names of their respective companies.

No investigation has been made of common-law trademark rights in any work.

**Table of Contents**

Section 1 Introduction to the VXI*plug&play* Systems Alliance and the IVI Foundation 1

Section 2 Overview of VISA Library Specification 1

2.1 Objectives of this Specification 1

2.2 Audience for this Specification 1

2.3 Scope and Organization of this Specification 2

2.4 Application of this Specification 2

2.5 References 3

2.6 Definition of Terms and Acronyms 4

2.7 Conventions 7

Section 3 VISA Resource Template 1

3.1 VISA Template Services 1

3.1.1 Control Services 1

3.1.2 Communication Services 3

3.2 VISA Template Interface Overview 4

3.2.1 VISA Template Attributes 4

3.2.2 VISA Template Operations 7

3.3 Lifecycle Services 8

3.3.1 Lifecycle Operations 8

3.3.1.1 **viClose**(vi) 9

3.4 Characteristic Control Services 10

3.4.1 Characteristic Control Operations 10

3.4.1.1 **viGetAttribute**(vi, attribute, attrState) 11

3.4.1.2 **viSetAttribute**(vi, attribute, attrState) 12

3.4.1.3 **viStatusDesc**(vi, status, desc) 14

3.5 Asynchronous Operation Control Services 15

3.5.1 Asynchronous Operation Control Operations 15

3.5.1.1 **viTerminate**(vi, degree, jobId) 16

3.6 Access Control Services 17

3.6.1 Session Access Control Service Model 17

3.6.1.1 Locking Mechanism 17

3.6.1.2 Lock Sharing 19

3.6.1.3 Access Privileges 19

3.6.1.4 Acquiring Exclusive Lock While Owning Shared Lock 22

3.6.1.5 Nested Locks 22

3.6.1.6 Locks on Remote Resources 22

3.6.2 Access Control Operations 23

3.6.2.1 **viLock**(vi, lockType, timeout, requestedKey, accessKey) 24

3.6.2.2 **viUnlock**(vi) 29

3.7 Event Services 31

3.7.1 Event Handling and Processing 31

3.7.1.1 Queuing Mechanism 32

3.7.1.2 Callback Mechanism 33

3.7.2 Exceptions 36

3.7.2.1 Exception Handling Model 36

3.7.2.2 Generating an Error Condition 37

3.7.2.3 VI\_EVENT\_EXCEPTION 38

3.7.3 Event Operations 38

3.7.3.1 **viEnableEvent**(vi, eventType, mechanism, context) 39

3.7.3.2 **viDisableEvent**(vi, eventType, mechanism) 42

3.7.3.3 **viDiscardEvents**(vi, eventType, mechanism) 44

3.7.3.4 **viWaitOnEvent**(vi, inEventType, timeout, outEventType, outContext) 46

3.7.3.5 **viInstallHandler**(vi, eventType, handler, userHandle) 49

3.7.3.6  **viUninstallHandler**(vi, eventType, handler, userHandle) 51

3.7.3.7  **viEventHandler**(vi, eventType, context, userHandle) 53

Section 4 VISA Resource Management 1

4.1 Organization of Resources 1

4.2 VISA Resource Manager Interface Overview 2

4.2.1 VISA Resource Manager Attributes 2

4.2.2 VISA Resource Manager Functions 2

4.2.3 VISA Resource Manager Operations 2

4.3 Access Services 3

4.3.1 Address String 3

4.3.1.1 Address String Grammar 3

4.3.2 System Configuration 7

4.3.3 Access Functions and Operations 8

4.3.3.1 **viOpenDefaultRM**(sesn) 9

4.3.3.2 **viOpen**(sesn, rsrcName, accessMode, timeout, vi) 11

4.3.3.3 **viParseRsrc**(sesn, rsrcName, intfType, intfNum) 14

4.3.3.4 **viParseRsrcEx**(sesn, rsrcName, intfType, intfNum, rsrcClass, unaliasedExpandedRsrcName, aliasIfExists) 16

4.4 Search Services 20

4.4.1 Resource Regular Expression 20

4.4.2 Search Operations 22

4.4.2.1 **viFindRsrc**(sesn, expr, findList, retcnt, instrDesc) 23

4.4.2.2 **viFindNext**(findList, instrDesc) 27

Section 5 VISA Resource Classes 28

5.1 Instrument Control Resource 30

5.1.1 INSTR Resource Overview 30

5.1.2 INSTR Resource Attributes 34

5.1.3 INSTR Resource Events 56

5.1.4 INSTR Resource Operations 62

5.1.5 Differences between VXI-11 and HiSLIP TCPIP INSTR Systems 64

5.2 Memory Access Resource 65

5.2.1 MEMACC Resource Overview 65

5.2.2 MEMACC Resource Attributes 67

5.2.3 MEMACC Resource Events 72

5.2.4 MEMACC Resource Operations 73

5.3 GPIB Bus Interface Resource 75

5.3.1 INTFC Resource Overview 75

5.3.2 INTFC Resource Attributes 76

5.3.3 INTFC Resource Events 80

5.3.4 INTFC Resource Operations 83

5.4 Mainframe Backplane Resource 84

5.4.1 BACKPLANE Resource Overview 84

5.4.2 BACKPLANE Resource Attributes 85

5.4.3 BACKPLANE Resource Events 89

5.4.4 BACKPLANE Resource Operations 91

5.5 Servant Device-Side Resource 92

5.5.1 SERVANT Resource Overview 92

5.5.2 SERVANT Resource Attributes 93

5.5.3 SERVANT Resource Events 97

5.5.4 SERVANT Resource Operations 100

5.6 TCP/IP Socket Resource 101

5.6.1 SOCKET Resource Overview 101

5.6.2 SOCKET Resource Attributes 101

5.6.3 SOCKET Resource Events 104

5.6.4 SOCKET Resource Operations 105

Section 6 VISA Resource-Specific Operations 1

6.1 Basic I/O Services 2

6.1.1 viRead(vi, buf, count, retCount) 2

6.1.2 viReadAsync(vi, buf, count, jobId) 5

6.1.3 viReadToFile(vi, fileName, count, retCount) 8

6.1.4 viWrite(vi, buf, count, retCount) 11

6.1.5 viWriteAsync(vi, buf, count, jobId) 13

6.1.6 viWriteFromFile(vi, fileName, count, retCount) 16

6.1.7 viAssertTrigger(vi, protocol) 18

6.1.8 viReadSTB(vi, status) 20

6.1.9 viClear(vi) 22

6.2 Formatted I/O Services 24

6.2.1 viSetBuf(vi, mask, size) 24

6.2.2 viFlush(vi, mask) 26

6.2.3 viPrintf(vi, writeFmt, arg1, arg2,...) 28

6.2.4 viVPrintf(vi, writeFmt, params) 37

6.2.5 viSPrintf(vi, buf, writeFmt, arg1, arg2, ...) 38

6.2.6 viVSPrintf(vi, buf, writeFmt, params) 39

6.2.7 viBufWrite(vi, buf, count, retCount) 41

6.2.8 viScanf(vi, readFmt, arg1, arg2,...) 43

6.2.9 viVScanf(vi, readFmt, params) 52

6.2.10 viSScanf(vi, buf, readFmt, arg1, arg2, ...) 52

6.2.11 viVSScanf(vi, buf, readFmt, params) 54

6.2.12 viBufRead(vi, buf, count, retCount) 55

6.2.13 viQueryf(vi, writeFmt, readFmt, arg1, arg2,...) 57

6.2.14 viVQueryf(vi, writeFmt, readFmt, params) 59

6.3 Memory I/O Services 61

6.3.1 viIn8(vi, space, offset, val8) 61

6.3.2 viIn16(vi, space, offset, val16) 61

6.3.3 viIn32(vi, space, offset, val32) 61

6.3.4 viIn64(vi, space, offset, val64) 61

6.3.5 viOut8(vi, space, offset, val8) 64

6.3.6 viOut16(vi, space, offset, val16) 64

6.3.7 viOut32(vi, space, offset, val32) 64

6.3.8 viOut64(vi, space, offset, val64) 64

6.3.9 viMoveIn8(vi, space, offset, length, buf8) 67

6.3.10 viMoveIn16(vi, space, offset, length, buf16) 67

6.3.11 viMoveIn32(vi, space, offset, length, buf32) 67

6.3.12 viMoveIn64(vi, space, offset, length, buf64) 67

6.3.13 viMoveIn8Ex(vi, space, offset64, length, buf8) 67

6.3.14 viMoveIn16Ex(vi, space, offset64, length, buf16) 67

6.3.15 viMoveIn32Ex(vi, space, offset64, length, buf32) 67

6.3.16 viMoveIn64Ex(vi, space, offset64, length, buf64) 67

6.3.17 viMoveOut8(vi, space, offset, length, buf8) 71

6.3.18 viMoveOut16(vi, space, offset, length, buf16) 71

6.3.19 viMoveOut32(vi, space, offset, length, buf32) 71

6.3.20 viMoveOut64(vi, space, offset, length, buf64) 71

6.3.21 viMoveOut8Ex(vi, space, offset64, length, buf8) 71

6.3.22 viMoveOut16Ex(vi, space, offset64, length, buf16) 71

6.3.23 viMoveOut32Ex(vi, space, offset64, length, buf32) 71

6.3.24 viMoveOut64Ex(vi, space, offset64, length, buf64) 71

6.3.25 viMove(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length) 75

6.3.26 viMoveEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length) 75

6.3.27 viMoveAsync(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId) 79

6.3.28 viMoveAsyncEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId) 79

6.3.29 viMapAddress(vi, mapSpace, mapBase, mapSize, access, suggested, address) 83

6.3.30 viMapAddressEx(vi, mapSpace, mapBase64, mapSize, access, suggested, address) 83

6.3.31 viUnmapAddress(vi) 86

6.3.32 viPeek8(vi, addr, val8) 87

6.3.33 viPeek16(vi, addr, val16) 87

6.3.34 viPeek32(vi, addr, val32) 87

6.3.35 viPeek64(vi, addr, val64) 87

6.3.36 viPoke8(vi, addr, val8) 88

6.3.37 viPoke16(vi, addr, val16) 88

6.3.38 viPoke32(vi, addr, val32) 88

6.3.39 viPoke64(vi, addr, val64) 88

6.4 Shared Memory Services 89

6.4.1 viMemAlloc(vi, size, offset) 89

6.4.2 viMemAllocEx(vi, size, offset64) 89

6.4.3 viMemFree(vi, offset) 91

6.4.4 viMemFreeEx(vi, offset64) 91

6.5 Interface Specific Services 92

6.5.1 viGpibControlREN(vi, mode) 92

6.5.2 viGpibControlATN(vi, mode) 94

6.5.3 viGpibSendIFC(vi) 96

6.5.4 viGpibCommand(vi, buf, count, retCount) 97

6.5.5 viGpibPassControl(vi, primAddr, secAddr) 99

6.5.6 viVxiCommandQuery(vi, mode, cmd, response) 100

6.5.7 viAssertIntrSignal(vi, mode, statusID) 102

6.5.8 viAssertUtilSignal(vi, line) 104

6.5.9 viMapTrigger(vi, trigSrc, trigDest, mode) 105

6.5.10 viUnmapTrigger(vi, trigSrc, trigDest) 108

6.5.11 viUsbControlOut (vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf) 110

6.5.12 viUsbControlIn (vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf, retCnt) 112

6.5.13 viPxiReserveTriggers (vi, cnt, trigBuses, trigLines, failureIndex) 114

Appendix A Required Attributes 1

A.1 Required Attribute Tables 1

Resource Template Attributes 1

INSTR Resource Attributes (Generic) (Continued) 2

INSTR Resource Attributes (Message Based) 2

INSTR Resource Attributes (GPIB and GPIB-VXI Specific) 2

INSTR Resource Attributes (VXI and GPIB-VXI Specific) 3

INSTR Resource Attributes (VXI and GPIB-VXI Specific) 3

INSTR Resource Attributes (ASRL Specific) 4

INSTR Resource Attributes (TCPIP Specific) 5

INSTR Resource Attributes (TCPIP Specific) 6

INSTR Resource Attributes (HiSLIP Specific) 6

INSTR Resource Attributes (VXI, GPIB-VXI, USB, and PXI Specific) 6

INSTR Resource Attributes (VXI, GPIB-VXI, and USB Specific) 6

INSTR Resource Attributes (USB Specific) 6

INSTR Resource Attributes (PXI Specific) 7

MEMACC Resource Attributes (Generic) 8

MEMACC Resource Attributes (VXI, GPIB-VXI, and PXI Specific) 10

MEMACC Resource Attributes (VXI and GPIB-VXI Specific) 10

MEMACC Resource Attributes (GPIB-VXI Specific) 11

INTFC Resource Attributes (Generic) 11

INTFC Resource Attributes (GPIB Specific) 12

BACKPLANE Resource Attributes (Generic) 12

BACKPLANE Resource Attributes (VXI and GPIB-VXI Specific) 13

SERVANT Resource Attributes (Generic) 13

SERVANT Resource Attributes (GPIB Specific) 14

SERVANT Resource Attributes (VXI Specific) 14

SERVANT Resource Attributes (TCPIP Specific) 14

SOCKET Resource Attributes (Generic) 15

SOCKET Resource Attributes (TCPIP Specific) 15

Appendix B Resource Summary Information 15

B.1 Summary of Attributes 15

B.2 Summary of Events 18

B.3 Summary of Operations 19

**Figures**

Figure 3.7.1 State Diagram for the Queuing Mechanism 3-32

Figure 3.7.2 State Diagram for the Callback Mechanism 3-35

**Tables**

Table 3.2.1 VISA Template Required Attributes 3-4

Table 3.2.2 ViVersion Description for VI\_ATTR\_RSRC\_IMPL\_VERSION and VI\_ATTR\_RSRC\_SPEC\_VERSION 3-5

Table 3.6.1 Types of Locks Acquired When Requesting Session Has No Lock 3-18

Table 3.6.2 Types of Locks Acquired When Requesting Session Has Exclusive Lock Only (Nesting) 3-18

Table 3.6.3 Types of Locks Acquired When Requesting Session Has Shared Lock (Nesting) 3-18

Table 3.6.4 Types of Locks Acquired When Requesting Session Has Shared and Exclusive Locks   
(Nesting) 3-18

Table 3.6.5 Current Session Has No Lock 3-20

Table 3.6.6 Current Session Has Exclusive Lock 3-20

Table 3.6.7 Current Session Has Shared Lock 3-20

Table 3.7.1 State Transitions for the Queuing Mechanism 3-33

Table 3.7.2 State Transition Table for the Callback Mechanism 3-36

Table 3.7.3 Special Values for eventType Parameter 3-40

Table 3.7.4 Special Values for mechanism Parameter 3-40

Table 3.7.5 Special Values for eventType Parameter 3-42

Table 3.7.6 Special Values for mechanism Parameter 3-43

Table 3.7.7 Special Values for eventType Parameter 3-45

Table 3.7.8 Special Values for mechanism Parameter 3-45

Table 3.7.9 Special Values for outEventType Parameter 3-47

Table 3.7.10 Special Values for outContext Parameter 3-47

Table 3.7.11 Special Values for handler Parameter 3-52

Table 4.3.1 Explanation of Address String Grammar 4-3

Table 4.3.2 Examples of Address Strings 4-6

Table 4.3.3 Special Values for rsrcClass Parameter 4-17

Table 4.3.4 Special Values for unaliasedExpandedRsrcName Parameter 4-17

Table 4.3.5 Special Values for aliasIfExists Parameter 4-18

Table 4.4.1 Special Characters 4-20

Table 4.4.2 Literals 4-20

Table 4.4.3 Regular Expression Characters and Operators 4-21

Table 4.4.4 Examples 4-21

Table 4.4.5 Special Values for findList Parameter 4-24

Table 4.4.6 Special Values for retcnt Parameter 4-24

Table 4.4.7 Special Characters and their Meaning 4-24

Table 4.4.8 Examples 4-25

Table 6.1.1 Special Values for retCount Parameter 6-3

Table 6.1.2 Special Values for jobId Parameter 6-6

Table 6.1.3 Special Values for retCount Parameter 6-9

Table 6.1.4 Special Values for retCount Parameter 6-12

Table 6.1.5 Special Values for jobId Parameter 6-14

Table 6.1.6 Special Values for retCount Parameter 6-17

Table 6.2.1 Special Values for retCount Parameter 6-42

Table 6.2.2 Special Values for retCount Parameter 6-56

Table 6.3.1 Special Values for jobId Parameter 6-80

Table 6.5.1 Special Values for mode Parameter 6-93

Table 6.5.2 Special Values for mode Parameter 6-95

Table 6.5.3 Special Values for retCount Parameter 6-98

Table 6.5.4 Special Values for mode Parameter 6-101

Table 6.5.5 Special Values for mode Parameter 6-103

Table 6.5.6 Special Values for trigSrc and trigDest Parameters 6-106

Table 6.5.7 Special Values for trigSrc Parameters 6-109

Table 6.5.8 Special Values for trigDest Parameters 6-109

Table 6.5.9 Special Values for retCnt Parameter 6-113

Section 1 Introduction to the VXI*plug&play* Systems Alliance and the IVI Foundation

The VXI*plug&play* Systems Alliance was founded by members who shared a common commitment to end-user success with open, multivendor VXI systems. The alliance accomplished major improvements in ease of use by endorsing and implementing common standards and practices in both hardware and software, beyond the scope of the VXIbus specifications. The alliance used both formal and de facto standards to define complete system frameworks. These standard frameworks gave end-users "plug & play" interoperability at both the hardware and system software level.

The IVI Foundation is an organization whose members share a common commitment to test system developer success through open, powerful, instrument control technology. The IVI Foundation’s primary purpose is to develop and promote specifications for programming test instruments that simplify interchangeability, provide better performance, and reduce the cost of program development and maintenance.

In 2002, the VXI*plug&play* Systems Alliance voted to become part of the IVI Foundation. In 2003, the VXI*plug&play* Systems Alliance formally merged into the IVI Foundation. The IVI Foundation has assumed control of the VXI*plug&play* specifications, and all ongoing work will be accomplished as part of the IVI Foundation.

All references to VXI*plug&play* Systems Alliance within this document, except contact information, were maintained to preserve the context of the original document.

Section 2 Overview of VISA Library Specification

This section introduces the VISA specification. The VISA specification is a document authored by the VXI*plug&play* Systems Alliance. The technical work embodied in this document and the writing of this document were performed by the VISA Technical Working Group.

This section provides a complete overview of the VISA specification, and gives readers general information that may be required to understand how to read, interpret, and implement individual aspects of this specification. This section is organized as follows:

• Objectives of this specification

• Audience for this specification

• Scope and organization of this specification

• Application of this specification

• References

• Definitions of terms and acronyms

• Conventions

• Communication

2.1 Objectives of this Specification

The VISA specification provides a common standard for the VXI*plug&play* System Alliance for developing multi-vendor software programs, including instrument drivers. This specification describes the VISA software model and the VISA Application Programming Interface (API).

VISA gives VXI and GPIB software developers, particularly instrument driver developers, the functionality needed by instrument drivers in an interface-independent fashion for MXI, embedded VXI, GPIB-VXI, GPIB, and asynchronous serial controllers. VXI*plug&play* drivers written to the VISA specifications can execute on VXI*plug&play* system frameworks that have the VISA I/O library.

2.2 Audience for this Specification

There are three audiences for this specification. The first audience is instrument driver developers—whether an instrument vendor, system integrator, or end user—who wish to implement instrument driver software that is compliant with the VXI*plug&play* standards. The second audience is I/O vendors who wish to implement VISA‑compliant I/O software. The third audience is instrumentation end users and application programmers who wish to implement applications that utilize instrument drivers compliant with this specification.

2.3 Scope and Organization of this Specification

This specification is organized in sections, with each section discussing a particular aspect of the VISA model.

Section 1 explains the VXI*plug&play* Systems Alliance and its relation to the IVI Foundation.

Section 2 provides an overview of this specification, including the objectives, scope and organization, application, references, definition of terms and acronyms, and conventions.

Section 3 describes the VISA Resource Template.

Section 4 describes the VISA Resource Manager Resource.

Section 5 presents the VISA Instrument Control Resource and other I/O resource classes.

Section 6 presents the operations defined in Section 5 and describes a compliant implementation.

2.4 Application of this Specification

This specification is intended for use by developers of VXI*plug&play* instrument drivers and by developers of VISA I/O software. It is also useful as a reference for end users of VXI*plug&play* instrument drivers. This specification is intended to be used in conjunction with the VPP-3.x specifications, including the *Instrument Drivers Architecture and Design Specification* (VPP-3.1), the *Instrument Driver Functional Body Specification* (VPP-3.2), the *Instrument Interactive Developer Interface Specification* (VPP-3.3), and the *Instrument Driver Programmatic Developer Interface Specification* (VPP-3.4). These related specifications describe the implementation details for specific instrument drivers that are used with specific system frameworks. VXI*plug&play* instrument drivers developed in accordance with these specifications can be used in a wide variety of higher-level software environments, as described in the *System* *Frameworks Specification* (VPP-2).

2.5 References

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

• ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation*

• ANSI/IEEE Standard 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands*

• ANSI/IEEE Standard 1014-1987, *IEEE Standard for a Versatile Backplane Bus: VMEbus*

• *ANSI/IEEE Standard 1174-2000, Standard Serial Interface for Programmable Instrumentation*

• PXI-4, PXI Module DescriptionFile Specification

• VPP-1, VXI*plug&play* Charter Document

• VPP-2, *System* *Frameworks Specification*

• VPP-3.1, *Instrument Drivers Architecture and Design Specification*

• VPP-3.2, *Instrument Functional Body Specification*

• VPP-3.3, *Instrument Driver Interactive Developer Interface Specification*

• VPP-3.4, *Instrument Driver Programmatic Developer Interface Specification*

• VPP-4.3.2, *VISA Implementation Specification for Textual Languages*

• VPP-4.3.3, *VISA Implementation Specification for the G Language*

• VPP-6, *Installation and Packaging Specification*

• VPP-7, *Soft Front Panel Specification*

• VPP-9, *Instrument Vendor Abbreviations*

• VXI-1, *VXIbus System Specification*, Revision 1.4, VXIbus Consortium

• VXI-11, *TCP/IP Instrument Protocol*, VXIbus Consortium

• IVI-6.1: High-Speed LAN Instrument Protocol (HiSLIP)

• IVI-6.3: IVI VISA PXI Plug-in

2.6 Definition of Terms and Acronyms

The following are some commonly used terms within this document

|  |  |
| --- | --- |
| **Address** | A string (or other language construct) that uniquely locates and identifies a resource. VISA defines an ASCII-based grammar that associates strings with particular physical devices or interfaces and VISA resources. |
| **ADE** | Application Development Environment |
| **API** | Application Programmers Interface. The direct interface that an end user sees when creating an application. The VISA API consists of the sum of all of the operations, attributes, and events of each of the VISA Resource Classes. |
| **Attribute** | A value within a resource that reflects a characteristic of the operational state of a resource. |
| **Bus Error** | An error that signals failed access to an address. Bus errors occur with low-level accesses to memory and usually involve hardware with bus mapping capabilities. For example, non-existent memory, a non-existent register, or an incorrect device access can cause a bus error. |
| **Commander** | A device that has the ability to control another device. This term can also denote the unique device that has sole control over another device (as with the VXI Commander/Servant hierarchy). |
| **Communication Channel** | The same as *Session*. A communication path between a software element and a resource. Every communication channel in VISA is unique. |
| **Controller** | A device that can control another device(s) or is in the process of performing an operation on another device. |
| **Device** | An entity that receives commands from a controller. A device can be an instrument, a computer (acting in a non-controller role), or a peripheral (such as a plotter or printer). In VISA, the concept of a device is generally the logical association of several VISA resources. |
| **HiSLIP** | HiSLIP (High Speed LAN Instrument Protocol) is a protocol for TCP-based instrument control that provides the instrument-like capabilities of conventional test and measurement protocols with minimal impact to performance. |
| **Instrument** | A device that accepts some form of stimulus to perform a designated task, test, or measurement function. Two common forms of stimuli are message passing and register reads and writes. Other forms include triggering or varying forms of asynchronous control. |
| **Interface** | A generic term that applies to the connection between devices and controllers. It includes the communication media and the device/controller hardware necessary for cross-communication. |
| **Instrument Driver** | Library of functions for controlling a specific instrument |
| **Mapping** | An operation that returns a reference to a specified section of an address space and makes the specified range of addresses accessible to the requester. This function is independent of memory allocation. |
| **Operation** | An action defined by a resource that can be performed on a resource. |
| **Process** | An operating system component that shares a system’s resources. A multi-process system is a computer system that allows multiple programs to execute simultaneously, each in a separate process environment. A single-process system is a computer system that allows only a single program to execute at a given point in time. |
| **Register** | An address location that either contains a value that is a function of the state of hardware or can be written into to cause hardware to perform a particular action or to enter a particular state. In other words, an address location that controls and/or monitors hardware. |
| **Resource Class** | The definition for how to create a particular resource. In general, this is synonymous with the connotation of the word *class* in object-oriented architectures. For VISA Instrument Control Resource Classes, this refers to the definition for how to create a resource that controls a particular capability of a device. |
| **Resource or Resource Instance** | In general, this term is synonymous with the connotation of the word *object* in object-oriented architectures. For VISA, *resource* more specifically refers to a particular implementation (or *instance* in object-oriented terms) of a Resource Class. In VISA, every defined software module is a resource. |
| **Session** | The same as *Communication Channel*. A communication path between a software element and a resource. Every communication channel in VISA is unique. |
| **SRQ** | IEEE 488 Service Request. This is an asynchronous request from a remote GPIB device that requires service. A service request is essentially an interrupt from a remote device. For GPIB, this amounts to asserting the SRQ line on the GPIB. For VXI, this amounts to sending the Request for Service True event (REQT). |
| **Status Byte** | A byte of information returned from a remote device that shows the current state and status of the device. If the device follows IEEE 488 conventions, bit 6 of the status byte indicates if the device is currently requesting service. |
| **Template Function** | Instrument driver subsystem function common to the majority of VXI*plug&play* instrument drivers |
| **Top-level Example** | A high-level test-oriented instrument driver function. It is typically developed from the instrument driver subsystem functions. |
| **Virtual Instrument** | A name given to the grouping of software modules (in this case, VISA resources with any associated or required hardware) to give the functionality of a traditional stand-alone instrument. Within VISA, a virtual instrument is the logical grouping of any of the VISA resources. The VISA Instrument Control Resources Organizer serves as a means to group any number of any type of VISA Instrument Control Resources within a VISA system. |
| **VISA** | Virtual Instrument Software Architecture. This is the general name given to this document and its associated architecture. The architecture consists of two main VISA components: the VISA Resource Manager and the VISA Instrument Control Resources. |
| **VISA Instrument Control Resources** | This is the name given to the part of VISA that defines all of the device-specific resource classes. VISA Instrument Control Resources encompass all defined device and interface capabilities for direct, low-level instrument control. |
| **VISA Resource Manager** | This is the name given to the part of VISA that manages resources. This management includes support for opening, closing, and finding resources; setting attributes, retrieving attributes, and generating events on resources; and so on. |
| **VISA Resource Template** | This is the name given to the part of VISA defines the basic constraints and interface definition for the creation and use of a VISA resource. All VISA resources must derive their interface from the definition of the VISA Resource Template. |

2.7 Conventions

Throughout this specification you will see the following headings on certain paragraphs. These headings instill special meaning on these paragraphs.

*Rules* must be followed to ensure compatibility with the System Framework. A rule is characterized by the use of the words **SHALL** and **SHALL NOT** in bold upper case characters. These words are not used in this manner for any other purpose other than stating rules.

*Recommendations* consist of advice to implementors that will affect the usability of the final device. They are included in this standard to draw attention to particular characteristics that the authors believe to be important to end user success.

*Permissions* are included to *authorize* specific implementations or uses of system components. A permission is characterized by the use of the word **MAY** in bold upper case characters. These permissions are granted to ensure specific System Framework components are well defined and can be tested for compatibility and interoperability.

*Observations* spell out implications of rules and bring attention to things that might otherwise be overlooked. They also give the rationale behind certain rules, so that the reader understands why the rule must be followed.

*A note on the text of the specification:* Any text that appears without heading should be considered as description of the standard and how the architecture was intended to operate. The purpose of this text is to give the reader a deeper understanding of the intentions of the specification including the underlying model and specific required features. As such, the implementor of this standard should take great care to ensure that a particular implementation does not conflict with the text of the standard.

Section 3 VISA Resource Template

VISA defines an architecture consisting of many resources that encapsulate device functionality. Each resource can give specialized services to applications or to other resources. Achieving this capability requires a high level of consistency in the operation of VISA resources. This level of consistency is achieved through a precisely defined, extensible interface, which provides a well-defined set of services. Each VISA resource derives its interface from a template that provides standard services for the resource. This increases the ability to reuse, test, and maintain the resource. These basic services from the template include the following:

• Creating and deleting sessions (Life Cycle Control)

• Modifying and retrieving individual resource characteristics called *Attributes* (Characteristic Control)

• Terminating queued operations (Asynchronous Operation Control)

• Restricting resource access (Access Control)

• Performing basic communication services (Operation Invocation and Event Reporting)

3.1 VISA Template Services

3.1.1 Control Services

The VISA template provides all the basic resource control services to applications. These basic services include controlling the life cycle of sessions to resources/devices and manipulating resource characteristics. A summary of these services for VISA is presented below:

**• Life Cycle Control**

VISA controls the life cycle of sessions, find lists, and events. Once an application has finished using any of them, it can use viClose() to free up all the system resources associated with it. The VISA system is also responsible for freeing up all associated system resources whenever an application becomes dysfunctional.

**• Characteristic Control**

Resources can have attributes associated with them. Some attributes depict the instantaneous state of the resource and some define alterable parameters to modify the behavior of the resources. VISA defines attribute manipulation operations to set and retrieve the status of resources. These attributes are defined by individual resources. The operation for modifying attributes is viSetAttribute() and the operation that retrieves the attributes is viGetAttribute().

**• Asynchronous Operation Control**

Resources can have asynchronous operations associated with them. These operations are invoked in the same way that all other operations are invoked. Instead of waiting for the actual job to be done, they register the job to be done and return immediately. When the I/O is complete, an event is generated to indicate the completion status of the associated operation. An application wanting to abort such an asynchronous operation can use viTerminate() with the unique job identifier returned from the operation to be aborted.

**• Access Control**

Applications can open multiple sessions to a VISA resource simultaneously. Applications can access the VISA resource through the different sessions concurrently. However, in certain cases, an application accessing a VISA resource might want to restrict other applications or sessions from accessing that resource. VISA defines a locking mechanism to restrict accesses to resources for such special circumstances. The operation used to acquire a lock on a resource is viLock(), and the operation to relinquish the lock is viUnlock().

3.1.2 Communication Services

Applications using VISA access resources by opening sessions to them. The primary method of communication to resources is by invoking operations. A VISA system also allows information exchange through events.

**• Operation Invocation**After establishing a session, an application can communicate with it by invoking operations associated with the resources. In VISA, every resource supports the operations described in the template. In addition to the specific error codes listed for each operation, the following generic error codes can be returned by any operation:

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given session does not support this operation. |
| VI\_ERROR\_NIMPL\_OPER | The given operation is not implemented. |
| VI\_ERROR\_SYSTEM\_ERROR | Unknown system error (miscellaneous error). |
| VI\_ERROR\_INV\_PARAMETER | The value of some parameter—which parameter is not known—is invalid. |
| VI\_ERROR\_USER\_BUF | A specified user buffer is not valid or cannot be accessed for the required size. |

**OBSERVATION 3.1.1**

It is possible that in the future, any operation may return success or error codes not listed in this specification. Therefore, it is important that applications check for general success or failure before comparing a return value to known return codes.

**OBSERVATION 3.1.2**

It is the intention of this specification to have success and warning codes be greater than or equal to zero and error codes less than zero. The specific status values are specified in the corresponding framework documents. Only unique identifiers are specified in this document.

**• Event Reporting**VISA provides callback, queuing, and waiting services that can inform sessions about resource-defined events.

**RECOMMENDATION 3.1.1**

If an operation defines an error code for a given parameter, a VISA implementation should normally use that error code.

**PERMISSION 3.1.1**

If a VISA implementation cannot determine which parameter caused an error, such as when using a lower-level driver, then it **MAY** return VI\_ERROR\_INV\_PARAMETER.

3.2 VISA Template Interface Overview

This section summarizes the interface that each VISA implementation must incorporate. The different attributes and operations are described in detail in subsequent sections.

3.2.1 VISA Template Attributes

**RULE 3.2.1**

Every VISA system **SHALL** implement the attributes and operations described in the VISA Resource Template.

**RULE 3.2.2**

Every VISA system **SHALL** implement the following attributes: VI\_ATTR\_RSRC\_NAME, VI\_ATTR\_RSRC\_SPEC\_VERSION, VI\_ATTR\_RSRC\_IMPL\_VERSION, VI\_ATTR\_RSRC\_MANF\_ID, VI\_ATTR\_RSRC\_MANF\_NAME, VI\_ATTR\_RM\_SESSION, VI\_ATTR\_USER\_DATA, VI\_ATTR\_MAX\_QUEUE\_LENGTH, VI\_ATTR\_RSRC\_CLASS, and VI\_ATTR\_RSRC\_LOCK\_STATE.

**RULE 3.2.3**

The value of the attribute VI\_ATTR\_RSRC\_SPEC\_VERSION **SHALL** be the value 00500400h.

**OBSERVATION 3.2.1**

The value of the attribute VI\_ATTR\_RSRC\_SPEC\_VERSION is a fixed value that reflects the version of the VISA specification to which the implementation is compliant. This value will change with subsequent versions of the specification.

Table 3.2.1 VISA Template Required Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_RSRC\_IMPL\_VERSION | RO | Global | ViVersion | 0h to FFFFFFFFh |
| VI\_ATTR\_RSRC\_LOCK\_STATE | RO | Global | ViAccessMode | VI\_NO\_LOCK  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 | 00500400h |
| VI\_ATTR\_RM\_SESSION | RO | Local | ViSession | N/A |
| VI\_ATTR\_MAX\_QUEUE\_LENGTH | R/W\* | Local | ViUInt32 | 1h to FFFFFFFFh |
| VI\_ATTR\_RSRC\_CLASS | RO | Global | ViString | N/A |
| VI\_ATTR\_USER\_DATA | R/W | Local | ViAddr | \*\* |
| VI\_ATTR\_USER\_DATA\_32 | R/W | Local | ViUInt32 | 0h to FFFFFFFFh |
| VI\_ATTR\_USER\_DATA\_64\*\*\* | R/W | Local | ViUInt64 | 0h to FFFFFFFFFFFFFFFFh |

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

\*\* Specified in the relevant VPP-4.3.*x* framework document.

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

**Attribute Descriptions**

VI\_ATTR\_RSRC\_IMPL\_VERSION Resource version that uniquely identifies each of the different revisions or implementations of a resource.

VI\_ATTR\_RSRC\_LOCK\_STATE The current locking state of the resource, reflecting any locks granted to an open session to the device using the same interface and protocol. The resource can be unlocked, locked with an exclusive lock, or locked with a shared lock.

VI\_ATTR\_RSRC\_MANF\_ID A value that corresponds to the VXI manufacturer ID of the manufacturer that created the implementation.

VI\_ATTR\_RSRC\_MANF\_NAME A string that corresponds to the VXI manufacturer name of the manufacturer that created the implementation.

VI\_ATTR\_RSRC\_NAME The unique identifier for a resource compliant with the address structure presented in Section 4.3.1, *Address String*.

VI\_ATTR\_RSRC\_SPEC\_VERSION Resource version that uniquely identifies the version of the VISA specification to which the implementation is compliant.

VI\_ATTR\_RM\_SESSION Specifies the session of the Resource Manager that was used to open this session.

VI\_ATTR\_MAX\_QUEUE\_LENGTH Specifies the maximum number of events that can be queued at any time on the given session.

VI\_ATTR\_RSRC\_CLASS Specifies the resource class (for example, “INSTR”) as defined in Section 5.

VI\_ATTR\_USER\_DATA Data used privately by the application for a particular session.

VI\_ATTR\_USER\_DATA\_32 This data is not used by VISA for any purposes and is

VI\_ATTR\_USER\_DATA\_64 provided to the application for its own use.

Table 3.2.2 ViVersion Description for VI\_ATTR\_RSRC\_IMPL\_VERSION and VI\_ATTR\_RSRC\_SPEC\_VERSION

|  |  |  |
| --- | --- | --- |
| **Bits 31 to 20** | **Bits 19 to 8** | **Bits 0 to 7** |
| Major Number | Minor Number | Sub-Minor Number |

**OBSERVATION 3.2.2**

VI\_ATTR\_RSRC\_LOCK\_STATE returns the combined lock state for all sessions of the same type. If there are three sessions open to the same device, with one being VXI-11 and two being HiSLIP sessions, then if one of the HiSLIP sessions holds a lock, both HiSLIP sessions will return a lock indication for this attribute, while the VXI-11 session will not.

**RULE 3.2.4**  
The value of the attribute VI\_ATTR\_RSRC\_IMPL\_VERSION **SHALL** increment with each new revision provided by a manufacturer.

**OBSERVATION 3.2.3**  
The value of the attribute VI\_ATTR\_RSRC\_IMPL\_VERSION is a value that is defined by the individual manufacturer with the only constraint of incrementing the total version value on subsequent revisions.

**RECOMMENDATION 3.2.1**  
It is recommended that the value of sub-minor versions be non-zero only for pre-release versions (beta). All officially released products should have a sub-minor value of zero.

**RULE 3.2.5**  
The attribute VI\_ATTR\_MAX\_QUEUE\_LENGTH **SHALL** be R/W (readable and writeable) until viEnableEvent() is called for the first time on a session.

**RULE 3.2.6**  
The attribute VI\_ATTR\_MAX\_QUEUE\_LENGTH **SHALL** be RO (read only and not writeable) after viEnableEvent() is called for the first time on a session.

**OBSERVATION 3.2.4**   
The previous two rules allow for a non-dynamically resizable implementation of queue lengths for VISA implementations. Queue lengths can be changed immediately after creation of a session but not after general operation has begun (that is, after viEnableEvent() has been called).

**RULE 3.2.7**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_USER\_DATA and VI\_ATTR\_USER\_DATA\_32 **SHALL** be identical.

**RULE 3.2.8**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_USER\_DATA and VI\_ATTR\_USER\_DATA\_64 **SHALL** be identical.

**RULE 3.2.9**

**IF** a framework is 32-bit, **THEN** the attribute VI\_ATTR\_USER\_DATA\_64 **SHALL NOT** be defined.

**OBSERVATION 3.2.5**

A user on a 32-bit framework can store 64-bit data via a private structure referenced by a 32-bit pointer.

**RULE 3.2.10**

**IF** a framework is 64-bit, **THEN** a VISA implementation **SHALL** provide only one user data value per session. **IF** a user calls viSetAttribute with the attribute VI\_ATTR\_USER\_DATA\_32 followed by a call to viGetAttribute with the attribute VI\_ATTR\_USER\_DATA\_64, **THEN** a VISA implementation **SHALL** return the 32-bit value that was previously set on that session.

3.2.2 VISA Template Operations

viClose(vi)

viGetAttribute(vi, attribute, attrState)

viSetAttribute(vi, attribute, attrState)

viStatusDesc(vi, status, desc)

viTerminate(vi, degree, jobId)

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

viUnlock(vi)

viEnableEvent(vi, eventType, mechanism, context)

viDisableEvent(vi, eventType, mechanism)

viDiscardEvents(vi, eventType, mechanism)

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

viInstallHandler(vi, eventType, handler, userHandle)

viUninstallHandler(vi, eventType, handler, userHandle)

**RULE 3.2.11**  
Every VISA system **SHALL** implement the following operations: viClose(), viGetAttribute(), viSetAttribute(), viStatusDesc(), viTerminate(), viLock(), viUnlock(), viEnableEvent(), viDisableEvent(), viDiscardEvents(), viWaitOnEvent(), viInstallHandler(), and viUninstallHandler().

3.3 Lifecycle Services

Once an application has opened a session to a VISA resource using some of the services in the VISA Resource Manager, it can use viClose() to close that session. The viClose() operation is also used to free find lists returned from the viFindRsrc() operation as well as events returned from the viWaitOnEvent() operation.

3.3.1 Lifecycle Operations

viClose(vi)

3.3.1.1 viClose(vi)

**Purpose**

Close the specified session, event, or find list.

**Parameter**

|  |  |  |  |
| --- | --- | --- | --- |
| **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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Session, event, or find list closed successfully. |
| VI\_WARN\_NULL\_OBJECT | The specified object reference is uninitialized. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_CLOSING\_FAILED | Unable to deallocate the previously allocated data structures corresponding to this session or object reference. |

**Description**

This operation closes a session, event, or a find list. In this process all the data structures that had been allocated for the specified vi are freed.

**Related Items**

See also viOpen().

**Implementation Requirements**

**RULE 3.3.1**

In a VISA system, a vi that receives the viClose() operation **SHALL** attempt to close the given vi and free all related data structures.

**RULE 3.3.2**

**IF** the value VI\_NULL is passed to the viClose() operation, **THEN** a VISA system **SHALL** return the completion code VI\_WARN\_NULL\_OBJECT.

3.4 Characteristic Control Services

Resources have attributes associated with them. Some attributes depict the instantaneous state of the resource and some define alterable parameters to modify behavior of the resources operations. VISA defines attribute manipulation operations to set and retrieve the status of resources. These attributes are defined by individual resources. This section describes the operations used to set and retrieve the value of individual attributes.

This section also includes an operation that can be used to retrieve a human-readable description for a given error code from a given session.

3.4.1 Characteristic Control Operations

viGetAttribute(vi, attribute, attrState)

viSetAttribute(vi, attribute, attrState)

viStatusDesc(vi, status, desc)

3.4.1.1 viGetAttribute(vi, attribute, attrState)

**Purpose**

Retrieve the state of an attribute.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession ViEvent ViFindList | Unique logical identifier to a session, event, or find list. |
| attribute | IN | ViAttr | Session, event, or find list attribute for which the state query is made. |
| attrState | OUT | ViAttrState | The state of the queried attribute for a specified resource. The interpretation of the returned value is defined by the individual resource. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Session, event, or find list attribute retrieved successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_ATTR | The specified attribute is not defined by the referenced session, event, or find list. |

**Description**

The viGetAttribute() operation is used to retrieve the state of an attribute for the specified session, event, or find list.

**Related Items**

See viSetAttribute().

**Implementation Requirements**

**RULE 3.4.1**

**IF** attribute is a string attribute, **THEN** viGetAttribute **SHALL** write no more than 256 characters into attrState, including the null character.

**OBSERVATION 3.4.1**

RULE 3.4.1 states the maximum length of a VISA string attribute to be 255 characters.

3.4.1.2 viSetAttribute(vi, attribute, attrState)

**Purpose**

Set the state of an attribute.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession ViEvent ViFindList | Unique logical identifier to a session, event, or find list. |
| attribute | IN | ViAttr | Session, event, or find list 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Attribute value set successfully. |
| VI\_WARN\_NSUP\_ATTR\_STATE | Although the specified attribute state is valid, it is not supported by this implementation. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_ATTR | The specified attribute is not defined by the referenced session, event, or find list. |
| VI\_ERROR\_NSUP\_ATTR\_STATE | The specified state of the attribute is not valid, or is not supported as defined by the session, event, or find list. |
| VI\_ERROR\_ATTR\_READONLY | The specified attribute is read-only. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |

**Description**

The viSetAttribute() operation is used to modify the state of an attribute for the specified session, event, or find list.

**Related Items**

See viGetAttribute().

**Implementation Requirements**

**RULE 3.4.2**

**IF** a resource cannot set an optional attribute state, **AND** the specified attribute state is valid, **AND** the attribute description does not specify otherwise, **THEN** the resource **SHALL** return the error code VI\_ERROR\_NSUP\_ATTR\_STATE.

**OBSERVATION 3.4.2**

Both VI\_WARN\_NSUP\_ATTR\_STATE and VI\_ERROR\_NSUP\_ATTR\_STATE indicate that the specified attribute state is not supported. Unless a specific rule states otherwise, a resource normally returns the error code VI\_ERROR\_NSUP\_ATTR\_STATE when it cannot set a specified attribute state. The completion code VI\_WARN\_NSUP\_ATTR\_STATE is intended to alert the application that although the specified optional attribute state is not supported, the application should not fail. One example is attempting to set an attribute value that would increase performance speeds. This is different than attempting to set an attribute value that specifies required but nonexistent hardware (such as specifying a VXI ECL trigger line when no hardware support exists) or a value that would change assumptions a resource might make about the way data is stored or formatted (such as byte order). See specific attribute descriptions for text that allows the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

**OBSERVATION 3.4.3**

The error code VI\_ERROR\_RSRC\_LOCKED is returned only if the specified attribute is Read/Write and Global, and the resource is locked by another session.

3.4.1.3 viStatusDesc(vi, status, desc)

**Purpose**

Return a user-readable description of the status code passed to the operation.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession ViEvent ViFindList | Unique logical identifier to a session, event, or find list. |
| status | IN | ViStatus | Status code to interpret. |
| desc | OUT | ViString | The user-readable string interpretation of the status code passed to the operation. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Description successfully returned. |
| VI\_WARN\_UNKNOWN\_STATUS | The status code passed to the operation could not be interpreted. |

**Description**

The viStatusDesc() operation is used to retrieve a user-readable string that describes the status code presented.

**Implementation Requirements**

**RULE 3.4.3**

**IF** a status code cannot be interpreted by the session, **THEN** the resource **SHALL** return the warning VI\_WARN\_UNKNOWN\_STATUS.

**RULE 3.4.4**

The output string desc **SHALL** be valid regardless of the status return value.

3.5 Asynchronous Operation Control Services

Resources can have asynchronous operations associated with them. These operations are invoked the same way in which all other operations are invoked. Instead of waiting for the actual job to be done, they register the job to be done and return immediately. An application that wants to abort such an asynchronous operation can use viTerminate() with the unique job identifier that is returned from the operation to be aborted. Examples of asynchronous operations are viReadAsync() and viWriteAsync(). Refer to Section 6, *VISA Resource-Specific Operations*, for more information on these and other asynchronous operations.

**PERMISSION 3.5.1**

A vendor **MAY** support multiple outstanding asynchronous operations per session.

**RULE 3.5.1**

**IF** an implementation supports multiple outstanding asynchronous operations per session **AND** the interface type of the resource is half duplex, **THEN** it **SHALL** process the operations in the order in which they are initiated.

**OBSERVATION 3.5.1**

For a full duplex resource such as asynchronous serial, write and read operations can occur in parallel without interfering with each other. For other resource types, processing asynchronous operations in the order in which they are initiated ensures that writes and reads happen in a predictable order.

**OBSERVATION 3.5.2**

This specification places no requirements on an implementation regarding the order of asynchronous operations with respect to synchronous operations on the same session, nor regarding the order of synchronous or asynchronous operations between sessions.

3.5.1 Asynchronous Operation Control Operations

viTerminate(vi, degree, jobId)

3.5.1.1 viTerminate(vi, degree, jobId)

**Purpose**

Request a VISA session to terminate normal execution of an 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Request serviced successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_JOB\_ID | Specified job identifier is invalid. |
| VI\_ERROR\_INV\_DEGREE | Specified degree is invalid. |

**Description**

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

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

**Related Items**

viReadAsync(), viWriteAsync(), viMoveAsync().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

3.6 Access Control Services

In VISA, applications can open multiple sessions to a VISA resource simultaneously. Applications can access the VISA resource through the different sessions concurrently. However, in certain cases, applications accessing a VISA resource might want to restrict other applications from accessing that resource. For example, suppose an application needs to perform successive write operations on a resource. The application also requires that during the sequence of writes, no other operation can be invoked through any other session to that resource. VISA defines a locking mechanism to restrict accesses to resources for such a special circumstance.

**RULE 3.6.1**

Every VISA resource on a multitasking or multithreading operating system **SHALL** safely handle concurrent operation invocations.

3.6.1 Session Access Control Service Model

3.6.1.1 Locking Mechanism

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 are serviced, or 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 execute certain operations will fail. Refer to descriptions of the individual operations to determine which would fail when a resource is locked. Locking a resource restricts access from other sessions, and in the case where an exclusive lock is acquired, guarantees that operations do not fail because other sessions have acquired a lock on that resource. Locking a resource prevents other sessions from acquiring an exclusive lock.

VISA defines two different types, or modes, of locks: *exclusive* and *shared* locks, which are denoted by VI\_EXCLUSIVE\_LOCK and VI\_SHARED\_LOCK, respectively. viLock() is used to acquire a lock on a resource, and viUnlock() is used to release the lock. This section describes the exclusive lock type. Section 3.6.1.2 describes shared locks, which are similar to exclusive locks in terms of access privileges, but which still can be shared between multiple sessions. The VI\_ATTR\_RSRC\_LOCK\_STATE attribute specifies the current locking state of the resource reflecting any lock granted to an open session to the device using the same interface and protocol.

**Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_RSRC\_LOCK\_STATE | RO | Global | ViAccessMode | VI\_NO\_LOCK  VI\_EXCLUSIVE\_LOCK  VI\_SHARED\_LOCK |

**RULE 3.6.2**

Every VISA resource **SHALL** support the VI\_ATTR\_RSRC\_LOCK\_STATE attribute.

**RULE 3.6.3**

Every VISA resource **SHALL** support both exclusive and shared locks.

Table 3.6.1 Types of Locks Acquired When Requesting Session Has No Lock

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Lock** | **Any Other Session Has** | | | |
| **Requested** | **No Locks** | **Exclusive Lock** | **Shared Lock** | **Shared and Exclusive Locks** |
| **Exclusive** | Yes | No | No | No |
| **Shared Lock** | Yes | No | Yes\* | Yes\* |

Table 3.6.2 Types of Locks Acquired When Requesting Session Has Exclusive Lock Only (Nesting)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Lock** | **Any Other Session Has** | | | |
| **Requested** | **No Locks** | **Exclusive Lock** | **Shared Lock** | **Shared and Exclusive Locks** |
| **Exclusive** | Yes | \*\* | \*\* | \*\* |
| **Shared Lock** | No | \*\* | \*\* | \*\* |

Table 3.6.3 Types of Locks Acquired When Requesting Session Has Shared Lock (Nesting)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Lock** | **Any Other Session Has** | | | |
| **Requested** | **No Locks** | **Exclusive Lock** | **Shared Lock** | **Shared and Exclusive Locks** |
| **Exclusive** | Yes | \*\* | Yes | No |
| **Shared Lock** | Yes | \*\* | Yes | Yes |

Table 3.6.4 Types of Locks Acquired When Requesting Session Has Shared and Exclusive Locks (Nesting)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Lock** | **Any Other Session Has** | | | |
| **Requested** | **No Locks** | **Exclusive Lock** | **Shared Lock** | **Shared and Exclusive Locks** |
| **Exclusive** | Yes | \*\* | Yes | \*\* |
| **Shared Lock** | No | \*\* | No | \*\* |

\* Only if the current session is aware of the access key. See Section 3.6.1.2, *Lock Sharing*, for more details.

\*\* The locking mechanism will not allow this situation to occur.

3.6.1.2 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, might have separate sessions to a resource and might want all the sessions in that application to have the same privilege as the session that locked the resource. In other cases, there might be a need to share locks among sessions in different applications. Essentially, sessions that acquired a lock to a resource may share the lock with other sessions it selects, and exclude access from other sessions.

This section discusses the mechanism that makes it possible to share locks. VISA defines a lock type—VI\_SHARED\_LOCK—that gives exclusive access privileges to a session along with the capability to share these exclusive privileges at the discretion of the original session. A session can lock a VISA resource using the lock type VI\_SHARED\_LOCK to get exclusive access privileges to the resource. When sharing the resource using a shared lock, the viLock() operation 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 the 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() operation. Invoking viLock() with the same key will register the new session to have the same access privilege as the original session. The session that acquired the access privileges through the sharing mechanism can also pass the access key to other sessions for sharing of resource. All the sessions sharing a resource using the shared lock should synchronize their accesses to maintain a consistent state of the resource.

VISA provides the flexibility for the applications to specify a key to use as the accessKey, instead of VISA generating the accessKey. The applications can suggest a key value to use through the requestedKey parameter of the viLock() operation. 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 resource was locked, the resource will grant the shared access to the session. If an application attempts to lock a resource using a shared lock, and passes VI\_NULL as the requestedKey parameter, then VISA will generate an accessKey for the session.

A session seeking to share an exclusive lock with other sessions needs to acquire a VI\_SHARED\_LOCK lock for this purpose. If it requests VI\_EXCLUSIVE\_LOCK, no valid access key will be returned. Consequently, the session will not be able to share it with any other sessions. This precaution minimizes the possibility of inadvertent or malicious access to the resource.

3.6.1.3 Access Privileges

If a session has an exclusive lock, other sessions cannot modify global attributes or invoke operations, but can still get attributes. If the session has a shared lock, other sessions that have shared locks can also modify global attributes and invoke operations. A session that does not have a shared lock will lack this capability.

If a session has a shared lock to a VISA resource, it can perform any operation and update any global attribute in that resource, unless some other session has an exclusive lock

The following tables describe the access privileges of a session under the various locking conditions.

Table 3.6.5 Current Session Has No Lock

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Access Privilege of Other Sessions** | | |
| **Operations Current Session Can Perform** | **All Other  Sessions Have  No Locks** | **One Session Has an Exclusive Lock** | **At Least One  Session Has a  Shared Lock** |
| **Get Attributes** | Yes | Yes | Yes |
| **Set Local Attributes** | Yes | Yes | Yes |
| **Set Global Attributes** | Yes | No | No |
| **Operations** | Yes | No\* | No\* |

Table 3.6.6 Current Session Has Exclusive Lock

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Access Privilege of Other Sessions** | | |
| **Operations Current Session Can Perform** | **All Other  Sessions Have  No Locks** | **One Session Has an Exclusive Lock\*\*** | **At Least One  Session Has a  Shared Lock** |
| **Get Attributes** | Yes | \*\* | Yes |
| **Set Local Attributes** | Yes | \*\* | Yes |
| **Set Global Attributes** | Yes | \*\* | Yes |
| **Operations** | Yes | \*\* | Yes |

Table 3.6.7 Current Session Has Shared Lock

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Access Privilege of Other Sessions** | | |
| **Operations Current Session Can Perform** | **All Other  Sessions Have  No Locks** | **One Session Has an Exclusive Lock\*\*\*** | **At Least One  Session Has a  Shared Lock** |
| **Get Attributes** | Yes | Yes\*\*\* | Yes |
| **Set Local Attributes** | Yes | Yes\*\*\* | Yes |
| **Set Global Attributes** | Yes | No\*\*\* | Yes |
| **Operations** | Yes | No\*, \*\*\* | Yes |

\* Some operations may be allowed. Refer to individual resources for more information.

\*\* These cases will not arise because the locking mechanism does not permit such locks to be granted to different sessions.

\*\*\* These cases arise when a session holding a shared lock also acquires an exclusive lock.

**OBSERVATION 3.6.1**

Tables 3.6.4, 3.6.5, and 3.6.6 list the general rules for what is permitted under various locking conditions. This information applies unless explicitly stated differently in specific descriptions of attributes or operations. However, there can be exceptions to the rule. For example, some operations may be permitted even when there is an exclusive lock on the 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 description of the individual operations and attributes.

In a VISA 2.2 system, only the I/O operations listed in Sections 5 and 6 are restricted by the locking scheme. Also, not all the operations are restricted by locking. Refer to descriptions of the individual operations to determine which are applicable for locking.

**RULE 3.6.4**

**IF** an operation respects locks **AND** the current session does not have the lock **AND** the locking session is not a HiSLIP session, **THEN** the operation **SHALL** immediately return VI\_ERROR\_RSRC\_LOCKED.

**RULE 3.6.5**

**IF** a session uses HiSLIP, **THEN** a VISA implementation **SHALL** pass exclusive and shared lock requests on that session to the device, excluding nested locks.

**RULE 3.6.6**

**IF** a session uses HiSLIP, **THEN** a VISA implementation **SHALL** return the HiSLIP remote lock state for VI\_ATTR RSRC\_LOCK\_STATE.

**RULE 3.6.7**

**IF** a lock is granted on a HiSLIP session, **THEN** operations that respect locks made by other HiSLIP sessions **SHALL** be blocked in the HiSLIP device until the lock is released and VISA **SHALL** return VI\_ERROR\_RSRC\_LOCKED.

**RECOMMENDATION 3.6.1**

For HiSLIP connections, VISA should wait its normal VISA timeout before returning VI\_ERROR\_RSRC\_LOCKED.

**OBSERVATION 3.6.2**

For HiSLIP sessions, access privileges are enforced by the HiSLIP device.

**RECOMMENDATION 3.6.2**

HiSLIP devices should extend HiSLIP lock enforcement to other connection styles. They should block operations that respect locks made by non-HiSLIP connections not holding the HiSLIP lock until that lock is released. VISA implementations cannot determine from a VISA resource descriptor which connections made via other interfaces or LAN protocols are to the same device as the one made via HiSLIP. Only the HiSLIP device knows which connections are to the same instrument or sub-instrument.

**OBSERVATION 3.6.3**

Holding HiSLIP locks and enforcing access privileges in the HiSLIP device allows multiple hosts to manage safe access to the HiSLIP device. Not returning immediate VI\_ERROR\_RSRC\_LOCKED errors allows more natural use of HiSLIP locks for critical-section-style programming patterns. HiSLIP locks may cause VI\_ERROR\_RSRC\_LOCKED errors after a VISA timeout if an operation is blocked by a lock.

3.6.1.4 Acquiring Exclusive Lock While Owning Shared Lock

When multiple sessions have acquired a shared lock, VISA allows one of the sessions to acquire an exclusive lock along with the shared lock it is holding. That is, a session holding a shared lock could also acquire an exclusive lock using the viLock() operation. The session holding both the exclusive and shared lock will have the same access privileges that it had when it was holding the shared lock only. However, this would preclude other sessions holding the shared lock from accessing the locked resource. When the session holding the exclusive lock unlocks the resource using the viUnlock() operation, all the sessions (including the one that had acquired the exclusive lock) will again have all the access privileges associated with the shared lock. This is useful when multiple sessions holding a shared lock must synchronize. This can also be used when one of the sessions must execute in a critical section. In the reverse case, in which a session is holding an exclusive lock only (no shared locks), VISA does not allow it to change to VI\_SHARED\_LOCK.

3.6.1.5 Nested Locks

VISA supports nested locking. That is, a session can lock the same VISA resource multiple times (for the same lock type). Unlocking the resource requires an equal number of invocations of the viUnlock() operation. Each session maintains a separate lock count for each type of locks. Repeated invocations of the viLock() operation for the same session will increase the appropriate lock count, depending on the type of lock requested. In the case of a shared lock, nesting viLock() calls will return with the same accessKey every time. In case of an exclusive lock, viLock() will not return any accessKey, regardless of whether it is nested or not. When a session locks the resource a multiple number of times, an equal number of invocations of the viUnlock() operation is required to actually unlock the resource. 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 can be actually unlocked only when the lock count is 0.

For nesting shared locks, VISA does not require an access key be passed in to invoke the viLock() operation. That is, a session does not need to pass in the access key obtained from the previous invocation of viLock() to gain a nested lock on the resource. However, if an application *does* pass in an access key when nesting on shared locks, it must be the correct one for that session. Refer to the description of the viLock() operation for further description of the accessKey parameter.

3.6.1.6 Locks on Remote Resources

The locking mechanism described in this section is guaranteed to work for all processes and resources existing on the same computer. When using remote resources, however, the networking protocol may not provide the ability to pass lock requests to the remote device or resource. In this case, locks should still behave as expected from multiple sessions on the same computer. For example, when using the VXI-11 protocol, exclusive lock requests can be sent to a device, but shared locks can only be handled locally. A less secure example is that multiple controllers in a VXI system may each have their own view of the system and may have duplicate locks without knowledge of each other.

**RULE 3.6.8**

A VISA implementation **SHALL** enforce locking as described in this specification for all sessions, processes, and resources on the same computer.

**RECOMMENDATION 3.6.3**

Multiple VISA entities on separate computers with access to the same resource should share lock information if possible.

3.6.2 Access Control Operations

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

viUnlock(vi)

3.6.2.1 viLock(vi, lockType, timeout, requestedKey, accessKey)

**Purpose**

Establish an access mode to the specified resource.

**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 locks). When trying to lock the resource as VI\_SHARED\_LOCK (shared), 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 description section below for more details. |
| accessKey | OUT | ViKeyId | This parameter should be set to VI\_NULL when lockType is VI\_EXCLUSIVE\_LOCK (exclusive locks). When trying to lock the resource as VI\_SHARED\_LOCK (shared), 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Specified access mode is successfully acquired. |
| VI\_SUCCESS\_NESTED\_EXCLUSIVE | Specified access mode is successfully acquired, and this session has nested exclusive locks. |
| VI\_SUCCESS\_NESTED\_SHARED | Specified access mode is successfully acquired, and this session has nested shared locks. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified 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 in is not a valid access key to the specified resource. |
| VI\_ERROR\_TMO | Specified type of lock could not be obtained within the specified timeout period. |

**Description**

This operation 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 operation 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 the 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 operation. To acquire nested locks, invoke the viLock() operation with the same lock type as the previous invocation of this operation. 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() operation 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.

**Related Items**

See viUnlock().

**Implementation Requirements**

**OBSERVATION 3.6.4**

It is the intention of this specification that ViKeyId be implemented as a string type. Since VI\_NULL may not be compatible with a string type in every language, a zero-length string can be substituted wherever VI\_NULL is used in a reference to a parameter of type ViKeyId.

**RULE 3.6.9**

A resource **SHALL** maintain an exclusive lock count and a shared lock count for each session that holds a lock on the resource.

**RULE 3.6.10**

**IF** a viLock() operation requests and acquires an exclusive lock successfully, **THEN** the exclusive lock count associated with that session **SHALL** be incremented by 1.

**RULE 3.6.11**

**IF** a viLock() operation requests and acquires an shared lock successfully, **THEN** the shared lock count associated with that session **SHALL** be incremented by 1.

**RULE 3.6.12**

**IF** a viLock() operation requesting a shared lock is invoked from a session whose associated exclusive lock count is non-zero (meaning the session has an exclusive lock) **THEN** the viLock() operation **SHALL** return the error VI\_ERROR\_RSRC\_LOCKED.

**RULE 3.6.13**

**IF** the lockType parameter is VI\_EXCLUSIVE\_LOCK, **THEN** the viLock() operation **SHALL** ignore the value of the requestedKey parameter.

**RULE 3.6.14**

**IF** the lockType parameter is VI\_EXCLUSIVE\_LOCK, **AND** the accessKey parameter points to a valid user buffer, **THEN** the viLock() operation **SHALL** set the value of accessKey to be a zero-length string.

**RULE 3.6.15**

**IF** an application makes a request for a shared lock on a resource **AND** the requestedKey value is set to VI\_NULL, **AND** the resource is not locked, **THEN** VISA **SHALL** generate the accessKey to allow sharing of the lock.

**OBSERVATION 3.6.5**

An accessKey used for sharing a lock to a resource need only be unique for a resource, but two different resources can have the same accessKey.

**RULE 3.6.16**

**IF** VISA generates the accessKey, **THEN** VISA **SHALL** generate the accessKey with a value that is guaranteed unique from all other VISA hosts.

**OBSERVATION 3.6.6**

An accessKey used for sharing a lock to a resource is guaranteed unique from other hosts if it is based in part on host-unique data, such as a GUID or MAC address.

**RULE 3.6.17**

**IF** an application makes a request for a shared lock on a resource, **AND** the requestedKey value is not set to VI\_NULL, **AND** the length of the requestedKey is greater than or equal to 256 characters, **THEN** the viLock() operation **SHALL** return VI\_ERROR\_INV\_ACCESS\_KEY.

**RULE 3.6.18**

**IF** an application makes a request for a shared lock on a resource, **AND** the requestedKey value is not set to VI\_NULL, **AND** the length of the requestedKey is less than 256 characters, **AND** the resource is not locked, **THEN** VISA **SHALL** use the requestedKey value as the access key to the resource.

**OBSERVATION 3.6.7**

An application can specify any valid string as a requestedKey value when acquiring a shared lock. Care should be taken in choosing the requestedKey value; otherwise, if a string is chosen that can be easily replicated, chances are other sessions may have chosen the same string and the sessions might unknowingly end up sharing the resource.

**RULE 3.6.19**

VISA **SHALL** support nested locking.

**RULE 3.6.20**

**IF** a session that holds a shared lock on the resource makes another invocation of the viLock() operation with the same lock type, **THEN** the resource **SHALL** return the same access key as the one returned in the previous invocation of viLock().

**RULE 3.6.21**

**IF** a session is being closed **AND** that session has lock(s) to the resource, **THEN** the resource locked through that session **SHALL** be unlocked by setting both exclusive and shared lock counts associated with that session to 0 before viClose() returns.

**RULE 3.6.22**

**IF** viLock() cannot acquire the lock immediately, **THEN** the operation **SHALL** wait for at least the time period specified in the timeout parameter before returning with an error.

**RULE 3.6.23**

**IF** the timeout is VI\_TMO\_IMMEDIATE **AND** viLock() cannot acquire the lock immediately, **THEN** the viLock() operation **SHALL** return immediately with an error.

**RULE 3.6.24**

**IF** a viLock() operation requests and acquires an exclusive lock successfully, **THEN** VISA **SHALL** ensure that the lock state of the resource associated with the given session is set to VI\_EXCLUSIVE\_LOCK.

**RULE 3.6.25**

**IF** a viLock() operation requests and acquires a shared lock successfully, **AND** the lock state of the resource associated with the given session was VI\_NO\_LOCK prior to the viLock() operation, **THEN** VISA **SHALL** ensure that the lock state of the resource associated with the given session is set to VI\_SHARED\_LOCK.

**RULE 3.6.26**

**IF** a viLock() operation requests and acquires a shared lock successfully, **AND** the lock state of the resource associated with the given session was not VI\_NO\_LOCK prior to the viLock() operation, **THEN** VISA **SHALL NOT** modify the lock state of the resource associated with the given session.

**RULE 3.6.27**

**IF** a viLock() operation requests and acquires an exclusive lock successfully, **AND** the exclusive lock count associated with the given session was zero prior to the viLock() operation, **THEN** viLock() **SHALL** return VI\_SUCCESS.

**RULE 3.6.28**

**IF** a viLock() operation requests and acquires an exclusive lock successfully, **AND** the exclusive lock count associated with the given session was non-zero prior to the viLock() operation, **THEN** viLock() **SHALL** return VI\_SUCCESS\_NESTED\_EXCLUSIVE.

**RULE 3.6.29**

**IF** a viLock() operation requests and acquires a shared lock successfully, **AND** the shared lock count associated with the given session was zero prior to the viLock() operation, **THEN** viLock() **SHALL** return VI\_SUCCESS.

**RULE 3.6.30**

**IF** a viLock() operation requests and acquires a shared lock successfully, **AND** the shared lock count associated with the given session was non-zero prior to the viLock() operation, **THEN** viLock() **SHALL** return VI\_SUCCESS\_NESTED\_SHARED.

**RULE 3.6.31**

**IF** a viLock() operation requests a shared lock, **AND** the exclusive lock count associated with the given session is zero, **AND** the shared lock count associated with the given session is non-zero, **AND** the requestedKey parameter is not set to VI\_NULL, **AND** the value of requestedKey is not the same as the access key for the resource associated with the given session, **THEN** viLock() **SHALL** return VI\_ERROR\_INV\_ACCESS\_KEY.

3.6.2.2 viUnlock(vi)

**Purpose**

Relinquish a lock for the specified resource.

**Parameter**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Lock successfully relinquished. |
| VI\_SUCCESS\_NESTED\_EXCLUSIVE | Call succeeded, but this session still has nested exclusive locks. |
| VI\_SUCCESS\_NESTED\_SHARED | Call succeeded, but this session still has nested shared locks. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_SESN\_NLOCKED | The current session did not have any lock on the resource. |

**Description**

This operation is used to relinquish the lock previously obtained using the viLock() operation.

**Related Items**

See viLock().

**Implementation Requirements**

**RULE 3.6.32**

**IF** the exclusive lock count is non-zero for the given session after an invocation of viUnlock(), **THEN** the operation **SHALL** return VI\_SUCCESS\_NESTED\_EXCLUSIVE.

**RULE 3.6.33**

**IF** the exclusive lock count is zero for the given session, **AND** the shared lock count is non-zero for the given session after an invocation of viUnlock(), **THEN** the operation **SHALL** return VI\_SUCCESS\_NESTED\_SHARED.

**RULE 3.6.34**

**IF** the exclusive lock count associated with a session is non-zero, **THEN** the exclusive lock count **SHALL** be decremented for each invocation of viUnlock() from that particular session.

**RULE 3.6.35**

**IF** the shared lock count associated with a session is non-zero, **AND** the exclusive lock count associated with the session is zero, **THEN** the shared lock count **SHALL** be decremented for each invocation of viUnlock() from that particular session.

**RULE 3.6.36**

When the exclusive lock count is decremented to 0 for a particular session, the session **SHALL** relinquish the exclusive lock on the resource.

**RULE 3.6.37**

When the shared lock count is decremented to 0 for a particular session, the session **SHALL** relinquish the shared lock on the resource.

**RULE 3.6.38**

**IF** both the exclusive and shared lock count associated with a session is 0, **THEN** any invocation of the viUnlock() operation on that session **shall not** decrement any lock count and **SHALL** return VI\_ERROR\_SESN\_NLOCKED.

**RULE 3.6.39**

A resource **SHALL** be unlocked only when the both the lock counts are 0 for all the sessions accessing the resource.

3.7 Event Services

VISA defines a common mechanism to notify an application when certain conditions occur. These conditions or occurrences are referred to as *events*. Typically, events occur because of a condition requiring the attention of applications. An event is a means of communication between a VISA resource and its applications.

VISA provides two independent mechanisms for an application to receive notification of event occurrences: queuing and callback handling. An application can enable either or both mechanisms using the viEnableEvent() operation. The callback handling mechanism can be enabled for one of two modes: immediate callback or delayed callback queuing. The viEnableEvent() operation is also used to switch between the two callback modes. The viDisableEvent() operation is used to disable either or both mechanisms.

In order to receive events using the queuing mechanism, an application must invoke the viWaitOnEvent() operation. In order to receive events using the callback mechanism, an application must install a callback handler using the viInstallHandler() operation.

When an application receives an event occurrence via either mechanism, it can determine information about the event by invoking viGetAttribute() on that event. When the application no longer needs the event information, it must call viClose() on that event.

3.7.1 Event Handling and Processing

The VISA event model provides two different ways for an application to receive event notification. The first method is to place all of the occurrences of a specified event type in a session-based queue. There is one event queue per event type per session. The application can receive the event occurrences later by dequeuing them with the viWaitOnEvent() operation. The other method is to call the application directly, invoking a function that the application installed prior to enabling the event. A callback handler is invoked on every occurrence of the specified event.

**RULE 3.7.1**

Every VISA resource **SHALL** implement both the queuing and callback event handling mechanisms.

The queuing and callback mechanisms are suitable for different programming styles. The queuing mechanism is generally useful for non-critical events that do not need immediate servicing. The callback mechanism is useful when immediate responses are needed. These mechanisms work independently of each other, so both can be enabled at the same time. By default, a session is not enabled to receive any events by either mechanism. The viEnableEvent() operation can be used to enable a session to respond to a specified event type using either the queuing mechanism, the callback mechanism, or both. Similarly, the viDisableEvent() operation can be used to disable one or both mechanisms. Because the two methods work independently of each other, one can be enabled or disabled regardless of the current state of the other.

The queuing mechanism is discussed in section 3.7.1.1, *Queuing Mechanism*. The callback mechanism is described in section 3.7.1.2, *Callback Mechanism*.

3.7.1.1 Queuing Mechanism

The queuing mechanism in VISA gives an application the flexibility to receive events only when it requests them. An application retrieves the event information by using the viWaitOnEvent() operation. If the specified event(s) exist in the queue, these operations retrieve the event information and return immediately. Otherwise, the application thread is blocked until the specified event(s) occur or until the timeout expires, whichever happens first. When an event occurrence unblocks a thread, the event is not queued for the session on which the wait operation was invoked. For more information about these operations, see section 3.7.2, *Event Operations.*

Figure 3.7.1 shows the state diagram for the queuing mechanism. This state diagram includes the enabling and disabling of the queuing mechanism and the corresponding operations.



Figure 3.7.1 State Diagram for the Queuing Mechanism

The queuing mechanism of a particular session can be in one of two different states: Disabled or Queuing (enabled for queuing). A session can transition between these two states using the viEnableEvent() or viDisableEvent() operation. Once a session is enabled for queuing (EQ transition to the Q state), all the event occurrences of the specified event type are queued. When a session is disabled for queuing (DQ transition to D state), any further event occurrences are not queued, but event occurrences that were already in the event queue are retained. The retained events can be dequeued at any time using the viWaitOnEvent() operation. An application can explicitly clear (flush) the event queue for a specified event type using the viDiscardEvents() operation.

**RULE 3.7.2**

**IF** there are any events in a session’s queue **AND** the queuing mechanism transitions between states, **THEN** the resource **SHALL NOT** discard any events from the queue.

The following table lists the state transitions and the corresponding values for the mechanism parameter in the viEnableEvent() and viDisableEvent() operations.

Table 3.7.1 State Transitions for the Queuing Mechanism

|  |  |  |  |
| --- | --- | --- | --- |
| **Destination State** | **Paths Leading to Destination State** | **Value of Mechanism Parameter** | **Operation to Use to Get State Transition** |
| Q | EQ | VI\_QUEUE | viEnableEvent() |
| D | DQ | VI\_QUEUE, VI\_ALL\_MECH | viDisableEvent() |

Every VISA resource provides an attribute for configuring and maintaining session queues. The VI\_ATTR\_MAX\_QUEUE\_LENGTH attribute specifies the maximum number of events that can be queued at any time on the given session.

**Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_MAX\_QUEUE\_LENGTH | R/W | Local | ViUInt32 | 1 to FFFFFFFFh |

**RULE 3.7.3**

Every VISA resource **SHALL** support the VI\_ATTR\_MAX\_QUEUE\_LENGTH attribute.

**RULE 3.7.4**

**IF** a queue is full **AND** a new event is to be placed on the queue, **THEN** the event with the lowest priority **SHALL** be discarded.

**RULE 3.7.5**

A VISA 2.2 system **SHALL** define the lowest priority to mean the most recent timestamp.

**OBSERVATION 3.7.1**

Because new events have a later timestamp (and therefore a lower priority) than events already on the queue, a queue full condition means that new events will be discarded and the state of the queue will not be altered.

3.7.1.2 Callback Mechanism

The VISA event model also allows applications to install functions that can be called back when a particular event type is received. The viInstallHandler() operation can be used to install handlers to receive specified event types. The handlers are invoked on every occurrence of the specified event, once the session is enabled for the callback mechanism. One handler must be installed before a session can be enabled for sensing using the callback mechanism.

**RULE 3.7.6**

**IF** no handler is installed for an event type **AND** an application calls viEnableEvent() **AND** the mechanism parameter is VI\_HNDLR, **THEN** the viEnableEvent() operation **SHALL** return the error VI\_ERROR\_HNDLR\_NINSTALLED.

VISA allows applications to install multiple handlers for an event type on the same session. Multiple handlers can be installed through multiple invocations of the viInstallHandler() operation, where each invocation adds to the previous list of handlers. If more than one handler is installed for an event type, each of the handlers is invoked on every occurrence of the specified event(s). VISA specifies that the handlers are invoked in Last In First Out (LIFO) order.

**RULE 3.7.7**

A VISA implementation **SHALL** allow at least 4 handlers to be installed on a given session for a given event type.

**PERMISSION 3.7.1**

A VISA implementation **MAY** allow as many handlers as it wishes. VISA does not enforce a maximum limit on the number of handlers that can be installed.

**RULE 3.7.8**

**IF** multiple handlers are installed for the same event type on the same session, **THEN** VISA **SHALL** invoke the handlers in the reverse order of their installation (LIFO order).

When a handler is invoked, the VISA resource provides the event context as a parameter to the handler. The event context is filled in by the resource. Applications can retrieve information from the event context object using the viGetAttribute() operation.

An application can supply a reference to any application-defined value while installing handlers. This reference is passed back to the application as the userHandle parameter to the callback routine during handler invocation. This allows applications to install the same handler with different application-defined contexts. For example, an application can install a handler with a fixed value 0x1 on a session for an event type. It can install the same handler with a different value, for example 0x2, for the same event type. The two installations of the same handler are different from one another. Both handlers are invoked when the event of the given type occurs. However, in one invocation the value passed to userHandle is 0x1 and in the other it is 0x2. Thus, event handlers are uniquely identified by a combination of the handler address and user context pair. This identification is particularly useful when different handling methods need to be done depending on the user context data. Refer to the viEventHandler() prototype for more information.

An application may install the same handler on multiple sessions. In this case, the handler is invoked in the context of each session for which it was installed (within the process environment).

**RULE 3.7.9**

**IF** a handler is installed on multiple sessions, **THEN** the handler **SHALL** be called once for each installation when an event occurs.

**OBSERVATION 3.7.2**

In a multithreaded operating system, the callback may occur in a different thread than the one from which viInstallHandler() is called.

**OBSERVATION 3.7.3**

The order of callbacks is only guaranteed for multiple handlers on a given session. A VISA implementation may perform callbacks to handlers on multiple sessions (or processes) in any order.

An application can uninstall any of the installed handlers using the viUninstallHandler() operation. This operation can also uninstall multiple handlers from the handler list at one time.

The following section discusses Figure 3.7.2, the state diagram of a resource implementing the callback mechanism. This state diagram includes the enabling and disabling of the callback mechanism in different modes. It also briefly describes the operations that can be used for state transitions. The table following the diagram lists different state transitions and parameter values for the viEnableEvent() and viDisableEvent() operations.



Figure 3.7.2 State Diagram for the Callback Mechanism

The callback mechanism of a particular session can be in one of three different states: Disabled, Handling, or suspended handling(Hbar). When a session transitions to the handling state (EH transition to H state), the callback handler is invoked for all the occurrences of the specified event type. When a session transitions to the suspended handling state (EHbar transition to Hbar), the callback handler is not invoked for any new event occurrences, but occurrences are kept in a suspended handler queue. The handler is invoked later, when a transition to the handling state occurs. When a session transitions to the disabled state (DH transition to the D state), the session is desensitized to any new event occurrences, but any pending occurrences are retained in the queue. In the suspended handling state, a maximum of the VI\_ATTR\_MAX\_QUEUE\_LENGTH number of event occurrences are kept pending. If the number of pending occurrences exceeds the value specified in this attribute, the lowest-priority events are discarded as described in section 3.7.1.1, *Queuing Mechanism*. An application can explicitly clear (flush) the callback queue for a specified event type using the viDiscardEvents() operation.

The following table lists the state transition diagram for the callback mechanism and the corresponding values for the mechanism parameter in the viEnableEvent() or viDisableEvent() operations.

Table 3.7.2 State Transition Table for the Callback Mechanism

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Destination State** | **Source State** | **Paths Leading to Destination State** | **Value of Mechanism Parameter** | **Operation to Use for State Transition** |
| H | D | EH | VI\_HNDLR | viEnableEvent() |
| H | Hbar | EH | VI\_HNDLR | viEnableEvent() |
| Hbar | D | EHbar | VI\_SUSPEND\_HNDLR | viEnableEvent() |
| Hbar | H | EHbar | VI\_SUSPEND\_HNDLR | viEnableEvent() |
| D | H | DH | VI\_HNDLR, VI\_SUSPEND\_HNDLR, VI\_ALL\_MECH | viDisableEvent() |
| D | Hbar | DH | VI\_SUSPEND\_HNDLR, VI\_HNDLR, VI\_ALL\_MECH | viDisableEvent() |

**RULE 3.7.10**

**IF** the callback mechanism mode for event handling is changed from VI\_SUSPEND\_HNDLR to VI\_HNDLR, **THEN** all the pending events for the event type specified in eventType parameter of viEnableEvent() **SHALL** be handled before viEnableEvent() completes.

**OBSERVATION 3.7.4**

The queuing mechanism and the callback mechanism operate independently of each other. In a VISA system, sessions keep information for event occurrences separate for both mechanisms. If one mechanism reaches its predefined limit for storing event occurrences, it does not directly affect the other mechanism.

3.7.2 Exceptions

In VISA, when an error occurs while executing an operation, the normal execution of a VISA resource halts. The resource notifies application of the error condition, invoking the application-specified callback routine for the exception event. The notification includes sufficient information for the application to know the cause of the error. Once notified, the application can tell the VISA system the action to take, depending on the severity of error. VISA provides this functionality through an exception event, which is referred to as an *exception* for the remainder of this document. The facility to handle exceptions is referred to as the *exception handling mechanism* in this document. In VISA, each error condition defined by operations of resources can cause exception events.

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

3.7.2.1 Exception Handling Model

This section describes the exception-handling model in VISA. In the VISA system, exceptions follow the event model presented earlier in this section. As described in the event-handling model, it is possible to install a callback handler which is invoked on an error. This installation can be done using the viInstallHandler() operation on a session. Once a handler is installed, a session can be enabled for exception event using viEnableEvent() operation.

When an error occurs for a session operation, the exception handler is executed synchronously; that is, the operation that caused the exception blocks until the exception handler completes its execution. When invoked, the exception handler can check the error condition and instruct the exception operation to take a specific action. It can instruct the exception operation to continue normally (returning the indicated error code) or to not invoke any additional handlers (in the case of handler nesting). A given implementation may choose to provide implementation-specific return codes for users’ exception handlers and may take alternate actions based on those implementation-specific codes.

**RULE 3.7.11**

All VISA implementations **SHALL** invoke exception handlers in the context of the thread that caused the exception event.

**PERMISSION 3.7.2**

A given implementation of VISA **MAY** define vendor-specific return codes for user exception handlers to return.

**PERMISSION 3.7.3**

A given implementation of VISA **MAY** take vendor-defined actions based on vendor-specific return codes from a user’s exception handler.

**OBSERVATION 3.7.5**

An example of a vendor-specific return code from an exception handler is one that causes the VISA implementation to close all sessions for the given process and exit the application. Remember that using vendor-specific return codes makes an application incompatible with other implementations.

As stated before, an exception operation blocks until the exception handler execution is completed. However, an exception handler sometimes may prefer to terminate the program prematurely without returning the control to the operation generating the exception. VISA does not preclude an application from using a platform-specific or language-specific exception handling mechanism from within the VISA exception handler. For example, the C++ try/catch block can be used in an application in conjunction with the C++ throw mechanism from within the VISA exception handler.

**OBSERVATION 3.7.6**

When using the C++ try/catch/throw or other exception-handling mechanisms, the control will not return to the VISA system. This has several important repercussions for both users and VISA implementors:

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

3.7.2.2 Generating an Error Condition

In VISA, when an error occurs, the normal execution of that session operation halts. The operation notifies the error condition to the application by raising an exception event. Raising the exception event will invoke the exception callback routine(s) installed for the particular session, based on whether this event is currently enabled for the given session.

One situation in which an exception event will not be generated is in the case of asynchronous operations. If the error is detected after the operation is posted (*i.e.*, once the asynchronous portion has begun), the status is returned normally via the I/O completion event. However, if an error occurs before the asynchronous portion begins (*i.e.*, the error is returned from the asynchronous operation itself), then the exception event will still be raised. This deviation is due to the fact that asynchronous operations already raise an event when they complete, and this I/O completion event may occur in the context of a separate thread previously unknown to the application. In summary, a single application event handler can easily handle error conditions arising from both exception events and failed asynchronous operations.

3.7.2.3 VI\_EVENT\_EXCEPTION

**Description**

Notification that an error condition has occurred during an operation invocation.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_EXCEPTION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS Status code returned by the operation generating the error.

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

**RULE 3.7.12**

The name of the operation contained in VI\_ATTR\_OPER\_NAME **SHALL** be exactly as presented in this specification, *The VISA Library*.

**OBSERVATION 3.7.7**

For an exception generated from the viLock() operation, VI\_ATTR\_OPER\_NAME would contain the string "viLock".

**OBSERVATION 3.7.8**

The intent of providing VI\_ATTR\_OPER\_NAME is to be able to provide diagnostic information, such as printing the name of the operation causing the event. Comparing the operation name in order to perform different actions, while valid, is not a recommended programming style.

3.7.3 Event Operations

viEnableEvent(vi, eventType, mechanism, context)

viDisableEvent(vi, eventType, mechanism)

viDiscardEvents(vi, eventType, mechanism)

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

viInstallHandler(vi, eventType, handler, userHandle)

viUninstallHandler(vi, eventType, handler, userHandle)

**Handler Prototype:**

viEventHandler(vi, eventType, context, userHandle)

3.7.3.1viEnableEvent(vi, eventType, mechanism, context)

**Purpose**

Enable notification of a specified event.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| eventType | IN | ViEventType | Logical event identifier. |
| mechanism | IN | ViUInt16 | Specifies event handling mechanisms to be enabled. The queuing mechanism is enabled by 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 |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Event enabled successfully. |
| VI\_SUCCESS\_EVENT\_EN | Specified event is already enabled for at least one of the specified mechanisms. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_INV\_MECH | Invalid mechanism specified. |
| VI\_ERROR\_INV\_CONTEXT | Specified event context is invalid. |
| VI\_ERROR\_HNDLR\_NINSTALLED | A handler is not currently installed for the specified event. The session cannot be enabled for the VI\_HNDLR mode of the callback mechanism. |
| VI\_ERROR\_NSUP\_MECH | The specified mechanism is not supported for the given event type. |

**Description**

This operation 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 operation replace the old callback mechanism with the new callback mechanism. Specifying VI\_ALL\_ENABLED\_EVENTS for the eventType parameter refers to all events that have previously been enabled on this session, making it easier to switch between the two callback mechanisms for multiple events.

Table 3.7.3 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. |

Table 3.7.4 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() operation. |
| 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. |

Notice that any combination of VISA-defined values for different parameters of the operation is also supported (except for VI\_HNDLR and VI\_SUSPEND\_HNDLR, which apply to different modes of the same mechanism).

**Related Items**

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

**Implementation Requirements**

**OBSERVATION 3.7.9**

This specification mandates that event queuing and callback mechanisms operate completely independently. As such, the enabling and disabling of the two modes in done independently (enabling one of the modes does not enable or disable the other mode). For example, if viEnableEvent() is called once with VI\_HNDLR and called a second time with VI\_QUEUE, both modes would be enabled.

**RULE 3.7.13**

**IF** viEnableEvent() is called with the mechanism parameter equal to the "bit-wise OR" of VI\_SUSPEND\_HNDLR and VI\_HNDLR, **THEN** viEnableEvent() **SHALL** return VI\_ERROR\_INV\_MECH.

**RULE 3.7.14**

**IF** the event handling mode is switched from VI\_SUSPEND\_HNDLR to VI\_HNDLR for an event type, **THEN** handlers that are installed for the event **SHALL** be called once for each occurrence of the corresponding event pending in the session (and dequeued from the suspend handler queue) before switching the modes.

**OBSERVATION 3.7.10**

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

**RULE 3.7.15**

**IF** the event handling mode is switched from VI\_HNDLR to VI\_SUSPEND\_HNDLR for an event type, **THEN** handler invocation for occurrences of the event type **SHALL** be deferred to a later time.

**RULE 3.7.16**

**IF** no handler is installed for an event type, **THEN** the request to enable the callback mechanism for the event type **SHALL** return VI\_ERROR\_HNDLR\_NINSTALLED.

**RULE 3.7.17**

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

3.7.3.2 viDisableEvent(vi, eventType, mechanism)

**Purpose**

Disable notification of an event type by the specified mechanisms.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| eventType | IN | ViEventType | Logical event identifier. |
| mechanism | IN | ViUInt16 | Specifies event handling mechanisms to be disabled. The queuing mechanism is disabled by specifying VI\_QUEUE, and 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. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Event disabled successfully. |
| VI\_SUCCESS\_EVENT\_DIS | Specified event is already disabled for at least one of the specified mechanisms. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_INV\_MECH | Invalid mechanism specified. |

**Description**

This operation 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.

Table 3.7.5 Special Values for eventType Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_ALL\_ENABLED\_EVENTS | Disable all events that were previously enabled. |

Table 3.7.6 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. |

Notice that any combination of VISA-defined values for different parameters of the operation is also supported.

**Related Items**

See the viEventHandler() prototype for its parameter description. Also see the viInstallHandler() and viUninstallHandler() descriptions for information about installing and uninstalling event handlers. Refer to event descriptions for context structure definitions.

**Implementation Requirements**

**RULE 3.7.18**

**IF** a request to disable an event handling mechanism is made for a session, **THEN** the events pending or queued in the session **SHALL** remain pending or queued, respectively, in the session.

**OBSERVATION 3.7.11**

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

3.7.3.3 viDiscardEvents(vi, eventType, mechanism)

**Purpose**

Discard event occurrences for specified event types and mechanisms in a session.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| eventType | IN | ViEventType | Logical event identifier. |
| mechanism | IN | ViUInt16 | Specifies the mechanisms for which the events are to be discarded. The VI\_QUEUE value is specified for the queuing mechanism and the VI\_SUSPEND\_HNDLR value is specified for the pending events in the callback mechanism. It is possible to specify both mechanisms simultaneously by specifying VI\_ALL\_MECH. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Event queue flushed successfully. |
| VI\_SUCCESS\_QUEUE\_EMPTY | Operation completed successfully, but queue was empty. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_INV\_MECH | Invalid mechanism specified. |

**Description**

This operation discards all pending occurrences of the specified event types and mechanisms from the specified session. The information about all the event occurrences that have not yet been handled is discarded. This operation is useful to remove event occurrences that an application no longer needs.

Table 3.7.7 Special Values for eventType Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_ALL\_ENABLED\_EVENTS | Discard events of every type that is enabled. |

Table 3.7.8 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. |

Notice that any combination of VISA-defined values for different parameters of the operation is also supported.

**Related Items**

Refer to the event handling mechanism.

**Implementation Requirements**

**OBSERVATION 3.7.12**

The event occurrences discarded by applications are not available to a session at a later time. This operation causes loss of event occurrences.

**OBSERVATION 3.7.13**

The viDiscardEvents() operation does not apply to event contexts that have already been delivered to the application.

3.7.3.4 viWaitOnEvent(vi, inEventType, timeout, outEventType, outContext)

**Purpose**

Wait for an occurrence of the specified event for a given session.

**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 | ViEventType | Logical identifier of the event actually received. |
| outContext | OUT | ViEvent | A handle specifying the unique occurrence of an event. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | 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 type specified by inEventType available for this session. |
| VI\_WARN\_QUEUE\_OVERFLOW | Wait terminated successfully on receipt of an event notification. There were more event occurrences of the type specified by inEventType than the configured queue size could hold, so the event queue overflowed. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_TMO | Specified event did not occur within the specified time period. |
| VI\_ERROR\_NENABLED | The session must be enabled for events of the specified type in order to receive them. |

**Description**

The viWaitOnEvent() operation suspends execution of a thread of application and waits for an event inEventType for a time period not to exceed that specified by timeout. Refer to individual event descriptions for context definitions. If the specified inEventType is VI\_ALL\_ENABLED\_EVENTS, the operation waits for any event that is enabled for the given session. If the specified timeout value is VI\_TMO\_INFINITE, the operation is suspended indefinitely.

Table 3.7.9 Special Values for outEventType Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the type of the event. |

Table 3.7.10 Special Values for outContext Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return an event context. |

**Related Items**

Refer to the overview of this section for more information on event handling. Also refer to the event descriptions in Section 5.

**Implementation Requirements**

**RULE 3.7.19**

**IF** the value VI\_TMO\_INFINITE is specified in the timeout parameter of viWaitOnEvent(), **THEN** the execution thread **SHALL** be suspended indefinitely to wait for an occurrence of an event.

**RULE 3.7.20**

**IF** the value VI\_TMO\_IMMEDIATE is specified in the timeout parameter of viWaitOnEvent(), **THEN** application execution **SHALL NOT** be suspended.

**OBSERVATION 3.7.14**

Notice that this operation can be used to dequeue events from an event queue by setting the timeout value to VI\_TMO\_IMMEDIATE.

**OBSERVATION 3.7.15**

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

**OBSERVATION 3.7.16**

A user of VISA must call viEnableEvent() to enable the reception of events of the specified type before calling viWaitOnEvent(). viWaitOnEvent() does not perform any enabling or disabling of event reception.

**RULE 3.7.21**

viWaitOnEvent() **SHALL** dequeue events pending in the queue regardless of the enabled state of reception of events.

**RULE 3.7.22**

**IF** the value VI\_NULL is specified in the outContext parameter of viWaitOnEvent(), **AND** the return value is successful, **THEN** the VISA system **SHALL** automatically invoke viClose() on the event context rather than returning it to the application.

**OBSERVATION 3.7.17**

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

**RULE 3.7.23**

**IF** a session has at least one event of the requested type in its queue, **AND** the requested event type has been disabled since the arrival of the last event, **THEN** calling viWaitOnEvent **SHALL** return a success code **AND** **SHALL NOT** return VI\_ERROR\_NENABLED.

3.7.3.5 viInstallHandler(vi, eventType, handler, userHandle)

**Purpose**

Install handlers for event callbacks.

**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 a client application. |
| userHandle | IN | ViAddr | A value specified by an application that can be used for identifying handlers uniquely for an event type. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Event handler installed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_INV\_HNDLR\_REF | The given handler reference is invalid. |
| VI\_ERROR\_HNDLR\_NINSTALLED | The handler was not installed. This may be returned if an application attempts to install multiple handlers for the same event on the same session. |

**Description**

This operation allows applications to install handlers on sessions. 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.

**Related Items**

See the viEventHandler() description for information.

**Implementation Requirements**

**RULE 3.7.24**

**IF** the value VI\_ANY\_HNDLR is passed as the handler parameter to viInstallHandler(), **THEN** the operation **SHALL** return the error VI\_ERROR\_INV\_HNDLR\_REF.

**RULE 3.7.25**

Every VISA implementation that returns a value greater than 00100100h for the VI\_ATTR\_RSRC\_SPEC\_VERSION attribute **SHALL** support multiple handlers per event type per session.

**OBSERVATION 3.7.18**

Previous versions of VISA (prior to Version 2.0) allowed only a single handler per event type per session.

3.7.3.6 viUninstallHandler(vi, eventType, handler, userHandle)

**Purpose**

Uninstall handlers for events.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Event handler successfully uninstalled. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_INV\_EVENT | Specified event type is not supported by the resource. |
| VI\_ERROR\_INV\_HNDLR\_REF | Either the specified handler reference or the user context value (or both) does not match any installed handler. |
| VI\_ERROR\_HNDLR\_NINSTALLED | A handler is not currently installed for the specified event. |

**Description**

This operation allows client applications to uninstall handlers for events on sessions. Applications should also specify the value in the userHandle parameter that was passed 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. The following tables list all the VISA-defined values and corresponding actions of uninstalling handlers.

Table 3.7.11 Special Values for handler Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_ANY\_HNDLR | Uninstall all the handlers with the matching value in the userHandle parameter. |

**Related Items**

See the viEventHandler() description for its parameter description. Also see the viEnableEvent() description for information about enabling different event handling mechanisms. Refer to individual event descriptions for context definitions.

**Implementation Requirements**

**RULE 3.7.26**

**If** no handler is installed for an event type as a result of this operation **and** a session is enabled for the callback mechanism in the VI\_HNDLR mode, **THEN** the callback mechanism for the event type **SHALL** be disabled for the session before this operation completes.

**OBSERVATION 3.7.19**

The userHandle value is used by the resource to uniquely identify the handlers along with the handler reference. Applications can use this value to process an event differently based on the value returned as a parameter of the handler.

3.7.3.7 viEventHandler(vi, eventType, context, userHandle)

**Purpose**

Event service handler procedure prototype.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Event handled successfully. |
| VI\_SUCCESS\_NCHAIN | Event handled successfully. Do not invoke any other handlers on this session for this event. |

**Description**

This user handle 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 event type 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.

**Related Items**

Refer to the overview of this section for more information on event handling and exception handling, and also to the event descriptions in Section 5.

**Implementation Requirements**

**RULE 3.7.27**

The VISA system **SHALL** automatically invoke the viClose() operation on the event context when a user handler returns.

**OBSERVATION 3.7.20**

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

**OBSERVATION 3.7.21**

If the user handler will not return to VISA, the application should call viClose() on the event context to manually delete the event object. This may occur when a handler throws a C++ exception in response to a VISA exception event. Note that this is an advanced case, so the previous observation applies in most cases.

**OBSERVATION 3.7.22**

Normally, an application should return VI\_SUCCESS from all callback handlers. If a specific handler does not want other handlers to be invoked for the given event for the given session, it should return VI\_SUCCESS\_NCHAIN. No return value from a handler on one session will affect callbacks on other sessions. Future versions of VISA (or specific implementations of VISA) may take actions based on other return values, so a user should return VI\_SUCCESS from handlers unless there is a specific reason to do otherwise.

Section 4 VISA Resource Management

This section describes the mechanisms available in VISA to control and manage resources. This includes, but is not limited to, the assignment of unique resource addresses, unique resource IDs, and operation invocation. Much of the work is done by the VISA Resource Manager.

The VISA Resource Manager is a resource like any other resource in the system. As such it derives its interface from the VISA Template. In addition, the VISA Resource Manager resource provides connectivity to all of the VISA resources registered with it. It gives applications control and access to individual resources and provides the services described as follows. The VISA Resource Manager relies on the resources available to it to service requests from the applications and other resources requiring service.

The VISA Resource Manager resource provides basic services to applications that include searching for resources, and the ability to open sessions to these resources. A summary of these services for VISA is presented below:

**• Access**

The VISA Resource Manager allows the opening of sessions to resources established on request by applications. Applications can request this service using viOpen(). The system has responsibility of freeing up all the associated system resources whenever an application closes the session or becomes dysfunctional.

**• Search**

These services are used to find a resource in order to establish a communication link to it. The search is based on a description string. Instead of locating and searching for individual resources, the VISA Resource Manager searches for resources associated with an interface. Applications can request this service by using the viFindRsrc() and viFindNext() operations.

4.1 Organization of Resources

The VISA Resource Manager provides access to all of the resources that are registered with it. It is therefore at the root of a subsystem of connected resources. Currently, one such entity is available by default to a VISA application after initialization—the Default Resource Manager. This identifier is used when opening resources, finding available resources, and performing other operations at the resource level.

**RULE 4.1.1**A VISA system **SHALL** make a Default Resource Manager resource available to the rest of the system.

**RULE 4.1.2**A session to the Default Resource Manager resource **SHALL** be returned from the viOpenDefaultRM() function.

4.2 VISA Resource Manager Interface Overview

This section summarizes the interface that each VISA implementation must incorporate. The different attributes and operations are described in detail in subsequent sections.

4.2.1 VISA Resource Manager Attributes

There are no attributes defined in the VISA Resource Manager resource in addition to those defined in the VISA Resource Template.

**RULE 4.2.1**The value of the attribute VI\_ATTR\_RSRC\_NAME for the Default Resource Manager **SHALL** be "", the empty string.

**RULE 4.2.2**The value of the attribute VI\_ATTR\_RM\_SESSION for the Default Resource Manager **SHALL** be VI\_NULL.

4.2.2 VISA Resource Manager Functions

viOpenDefaultRM(sesn)

**RULE 4.2.3**  
Every VISA Resource Manager resource **SHALL** implement the following function: viOpenDefaultRM().

4.2.3 VISA Resource Manager Operations

viFindRsrc(sesn, expr, findList, retcnt, instrDesc)

viFindNext(findList, instrDesc)

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

viParseRsrc(sesn, rsrcName, intfType, intfNum)

viParseRsrcEx(sesn, rsrcName, intfType, intfNum, rsrcClass, unaliasedExpandedRsrcName, aliasIfExists)

**RULE 4.2.4**  
Every VISA Resource Manager resource **SHALL** implement the following operations: viFindRsrc(), viFindNext(), viOpen(), viParseRsrc(), and viParseRsrcEx().

4.3 Access Services

The VISA Resource Manager provides facilities to create sessions to resources. viOpenDefaultRM() is used by an application to get access to the default Resource Manager. viOpen() is used to get access to a resource through a session. In order to open a session to a device resource or any other type of resource with VISA, it is essential to be able to uniquely identify a resource in the system. The Address String defined in the following section is the mechanism by which the resource must be uniquely identified.

4.3.1 Address String

An address string must uniquely identify a VISA resource. The address string is used in viOpen().

4.3.1.1 Address String Grammar

The grammar for the Address String is shown in Table 4.3.1. Optional string segments are shown in square brackets ([]).

Table 4.3.1 Explanation of Address String Grammar

|  |  |
| --- | --- |
| **Interface** | **Grammar** |
| VXI | VXI[*board*]::*VXI logical address*[::INSTR] |
| VXI | VXI[*board*]::MEMACC |
| VXI | VXI[*board*][::*VXI logical address*]::BACKPLANE |
| VXI | VXI[*board*]::SERVANT |
| GPIB-VXI | GPIB-VXI[*board*]::*VXI logical address*[::INSTR] |
| GPIB-VXI | GPIB-VXI[*board*]::MEMACC |
| GPIB-VXI | GPIB-VXI[*board*][::*VXI logical address*]::BACKPLANE |
| GPIB | GPIB[*board*]::*primary address*[::*secondary address*][::INSTR] |
| GPIB | GPIB[*board*]::INTFC |
| GPIB | GPIB[*board*]::SERVANT |
| ASRL | ASRL[*board*][::INSTR] |
| TCPIP | TCPIP[*board*][::*LAN device name*]::SERVANT |
| TCPIP | TCPIP[*board*]::*host address*[::*LAN device name*][::INSTR] |
| TCPIP | TCPIP[board]::*host address*[::*HiSLIP device name*[,*HiSLIP* *port*]][::INSTR] |
| TCPIP | TCPIP[*board*]::*host address*::*port*::SOCKET |
| USB | USB[*board*]::*manufacturer ID*::*model code*::*serial number*[::*USB interface number*][::INSTR] |
| PXI | PXI[*bus*]::*device*[::*function*][::INSTR] |
| PXI | PXI[*interface*]::*bus*-*device*[.*function*][::INSTR] |
| PXI | PXI[*interface*]::CHASSIS*chassis*::SLOT*slot*[::FUNC*function*][::INSTR] |
| PXI | PXI[*interface*]::MEMACC |
| PXI | PXI[*interface*]::*chassis number*::BACKPLANE |

The VXI keyword is used for VXI instruments via either embedded or MXIbus controllers. The GPIB-VXI keyword is used for a GPIB-VXI controller. The GPIB keyword can be used to establish communication with a GPIB device. The ASRL keyword is used to establish communication with an asynchronous serial (such as RS-232) device. The TCPIP keyword is used to establish communication with Ethernet instruments. The USB keyword is used to establish communication with USB instruments.

Resources classes, including INSTR (instrument control), are discussed in Section 5.

In the PXI INSTR strings, the *bus*, *device*, and *function* parameters refer to the PCI bus number, PCI device number, and PCI function number that would be used to access the resource in PCI configuration space. The *chassis* and *slot* parameters correspond to the chassis number and slot number attributes of the resource.

Notice that the address string for a PXI INSTR resource has three acceptable formats.

The default value for optional string segments is shown below.

|  |  |
| --- | --- |
| **Optional String Segment** | **Default Value** |
| board | 0 |
| GPIB secondary address | none |
| LAN device name | inst0 |
| HiSLIP device name | hislip0 |
| HiSLIP port | 4880 |
| USB interface number | lowest numbered relevant interface |
| PCI function number | 0 |

**RULE 4.3.1**

The VISA resource string for a USB INSTR **SHALL** use hexadecimal digits for the manufacturer ID and model code. Specifically, the new variables must be present in “0xXXXX” format.

**RULE 4.3.2**

In a system where all PCI devices are accessible through a single configuration address space, the *interface* parameter **SHALL** be zero (0) for all resources.

**RULE 4.3.3**

A VISA implementation that supports PXI INSTR resources **SHALL** support all defined PXI INSTR string formats.

**OBSERVATION 4.3.1**

The VISA resource string for a single-function device on bus zero (0) is identical in both formats for PXI INSTR resources.

**OBSERVATION 4.3.2**

The Bus/device/function legacy string format does not allow for multiple PXI systems with separate address spaces. Although PCI-based systems typically have a single address space today, there may be a need for multiple address spaces in the future.

**RULE 4.3.4**

A VISA implementation **SHALL** support a hostname or a dot-delimited IPv4 IP address forTCPIP *host address*.

**RULE 4.3.5**

A VISA implementation **SHALL** support a http URI host address forTCPIP *host address* for expressing an IPv6 IP address in a HiSLIP VISA address strings.

**OBSERVATION 4.3.3**

Http URI host address formats are specified in IETF RFC3986, Section3.2.2. For IPv4 IP addresses, they are simply four dot-delimited decimal numbers. For IPv6 IP addresses, the address string is enclosed in square brackets and can contain ‘::’ character strings (example: [fe80::1]). Hostnames are handled as simple strings. This RFC makes provision for future versions of IP addresses as well.

**RECOMMENDATION 4.3.1**

A VISA implementation should accept a http URI address forTCPIP *host address* including IPv6 IP addresses inside square brackets for other TCPIP non-HiSLIP address strings. Returning VI\_RSRC\_NSUP\_OPER is acceptable in this case.

**RULE 4.3.6**

A VISA implementation **SHALL** connect via HiSLIP for address strings with an alphanumeric *HiSLIP device name* starting with ‘hislip’.

**RULE 4.3.7**

A VISA implementation **SHALL** connect via VXI-11 for address strings with an alphanumeric *LAN device name* starting with ‘vxi’ for VXI-11.1, ‘gpib’ for VXI-11.2, and ‘inst’ for VXI-11.3. [See the VXI-11 specification documents for details.]

**RULE 4.3.8**

**IF** the device name is omitted **AND** the device supports VXI-11 **AND** the host address indicates an IPv4 connection, **THEN** VISA **SHALL** connect using the VXI-11 protocol.

Table 4.3.2 Examples of Address Strings

|  |  |
| --- | --- |
| **Address String** | **Description** |
| VXI0::1::INSTR | A VXI device at logical address 1 in VXI interface VXI0. |
| GPIB-VXI::9::INSTR | A VXI device at logical address 9 in a GPIB-VXI controlled VXI system. |
| GPIB::1::0::INSTR | A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0. |
| ASRL1::INSTR | A serial device located on port 1. |
| VXI::MEMACC | Board-level register access to the VXI interface. |
| GPIB-VXI1::MEMACC | Board-level register access to GPIB-VXI interface number 1. |
| GPIB2::INTFC | Interface or raw resource for GPIB interface 2. |
| VXI::1::BACKPLANE | Mainframe resource for chassis 1 on the default VXI system, which is interface 0. |
| GPIB-VXI2::BACKPLANE | Mainframe resource for default chassis on  GPIB-VXI interface 2. |
| GPIB1::SERVANT | Servant/device-side resource for GPIB interface 1. |
| VXI0::SERVANT | Servant/device-side resource for VXI interface 0. |
| TCPIP0::1.2.3.4::999 ::SOCKET | Raw TCP/IP access to port 999 at the specified address. |
| TCPIP::devicename. company.com::INSTR | A TCP/IP device using VXI-11 located at the specified address. This uses the default LAN Device Name of inst0. |
| TCPIP::1.2.3.4::inst0 ::INSTR | A TCP/IP device using VXI-11 located at IPv4 IP address 1.2.3.4. |
| TCPIP::[fe80::1] ::hislip0::INSTR | A TCP/IP device using HiSLIP located at IPv6 IP address fe80::1. |
| USB::0x1234::0x5678 ::A22-5::INSTR | A USB Test & Measurement class device with manufacturer ID 0x1234, model code 0x5678, and serial number A22-5. This uses the device’s first available USBTMC interface. This is usually number 0. |
| PXI0::3-18::INSTR | PXI device 18 on bus 3. |
| PXI0::3-18.2::INSTR | Function 2 on PXI device 18 on bus 3. |
| PXI0::21::INSTR | PXI device 21 on bus 0. |
| PXI0::CHASSIS1::SLOT4 ::INSTR | PXI device in slot 4 of chassis 1. |
| PXI0::MEMACC | Access to system controller memory available to devices in the PXI system. |
| PXI0::1::BACKPLANE | Mainframe resource for PXI chassis 1. |

4.3.2 System Configuration

Although the VISA specification describes certain default values for an implementation, it is valid for a VISA implementation to allow a user to change various settings on a system via some external configuration utility. Such a utility is neither defined nor mandated by this VISA specification. Several optional return values are defined by the VISA Resource Manager, but these may not apply to all VISA implementations.

**PERMISSION 4.3.1**

A VISA implementation **MAY** provide an external configuration utility.

**RULE 4.3.9**

A VISA implementation that supports PXI INSTR resources **SHALL** provide a tool for registering modules using the module.ini files specified in the PXI Software Specification. The tool **SHALL** provide a mechanism for registering those devices in a programmatic or scriptable manner.

**RECOMMENDATION 4.3.2**

A VISA implementation that supports PXI INSTR resources should provide an interactive tool for registering modules that does not require a module.ini file.

**OBSERVATION 4.3.4**

PXI end users will first install VISA, then use tools provided with the VISA implementation to register the module description file with the operating system, then install the hardware. For example, on Microsoft Windows operating systems, VISA would read the module description and generate a Windows Setup Information (.inf) file that the operating system would then use to identify the hardware. Installing the software before the hardware ensures that the information in the module description file is available to the operating system when it needs to identify the hardware.

4.3.3 Access Functions and Operations

viOpenDefaultRM(sesn)

viOpen(sesn, rsrcName, accessMode, timeout, sesn)

viParseRsrc(sesn, rsrcName, intfType, intfNum)

viParseRsrcEx(sesn, rsrcName, intfType, intfNum, rsrcClass,

unaliasedExpandedRsrcName, aliasIfExists)

4.3.3.1 viOpenDefaultRM(sesn)

**Purpose**

Return a session to the Default Resource Manager resource.

**Parameter**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Session to the Default Resource Manager resource created successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_SYSTEM\_ERROR | The VISA system failed to initialize. |
| VI\_ERROR\_ALLOC | Insufficient system resources to create a session to the Default Resource Manager resource. |
| VI\_ERROR\_INV\_SETUP | Some implementation-specific configuration file is corrupt or does not exist. |
| VI\_ERROR\_LIBRARY\_NFOUND | A code library required by VISA could not be located or loaded. |

**Description**

This function must be called before any VISA operations 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.

**Related Items**

See also viOpen(), viFindRsrc().

**Implementation Requirements**

**RULE 4.3.10**

The viOpenDefaultRM() function **SHALL** be invoked before any operation in VISA.

**RULE 4.3.11**

Repetitive calls to the viOpenDefaultRM() function **SHALL** return new and unique sessions to the Default Resource Manager.

**RULE 4.3.12**

**IF** the viClose() operation is invoked on a session returned from viOpenDefaultRM(), **THEN** all VISA sessions opened with the corresponding Default Resource Manager session **SHALL** be closed.

**RULE 4.3.13**

**IF** the viClose() operation is invoked on a session returned from viOpenDefaultRM(), **THEN** all VISA system resources associated with the corresponding Default Resource Manager session **SHALL** be deallocated.

**RULE 4.3.14**

For compatibility with earlier versions of this specification, a VISA system **SHALL** provide the function viGetDefaultRM() with the same signature and semantics as viOpenDefaultRM().

**OBSERVATION 4.3.5**

The function viOpenDefaultRM() renders the viGetDefaultRM() function obsolete. The function name has changed to match the semantics of the action that the function performs.

4.3.3.2 viOpen(sesn, rsrcName, accessMode, timeout, vi)

**Purpose**

Open a session to the specified 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. |
| accessMode | IN | ViAccessMode | Specifies the modes by which the resource is to be accessed. The value VI\_EXCLUSIVE\_LOCK is used to acquire an exclusive lock immediately upon opening a session; if a lock cannot be acquired, the session is closed and an error is returned. The value VI\_LOAD\_CONFIG is used to configure attributes to values specified by some external configuration utility; if this value is not used, the session uses the default values provided by this specification. Multiple access modes can be used simultaneously by specifying a "bit-wise OR" of the above values. |
| timeout | IN | ViUInt32 | If the accessMode parameter requests a lock, then this parameter specifies the absolute time period (in milliseconds) that the resource waits to get unlocked before this operation returns an error. |
| vi | OUT | ViSession | Unique logical identifier reference to a session. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Session opened successfully. |
| VI\_SUCCESS\_DEV\_NPRESENT | Session opened successfully, but the device at the specified address is not responding. |
| VI\_WARN\_CONFIG\_NLOADED | The specified configuration either does not exist or could not be loaded; using VISA-specified defaults. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given sesn does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session. |
| VI\_ERROR\_INV\_RSRC\_NAME | Invalid resource reference specified. Parsing error. |
| VI\_ERROR\_INV\_ACC\_MODE | Invalid access mode. |
| VI\_ERROR\_RSRC\_NFOUND | Insufficient location information or resource not present in the system. |
| VI\_ERROR\_ALLOC | Insufficient system resources to open a session. |
| VI\_ERROR\_RSRC\_BUSY | The resource is valid, but VISA cannot currently access it. |
| VI\_ERROR\_RSRC\_LOCKED | Specified type of lock cannot be obtained because the resource is already locked with a lock type incompatible with the lock requested. |
| VI\_ERROR\_TMO | A session to the resource could not be obtained within the specified timeout period. |
| VI\_ERROR\_LIBRARY\_NFOUND | A code library required by VISA could not be located or loaded. |
| VI\_ERROR\_INTF\_NUM\_NCONFIG | The interface type is valid but the specified interface number is not configured. |

**Description**

This operation opens a session to the specified device. It returns a session identifier that can be used to call any other operations of that device.

**Related Items**

See also viClose().

**Implementation Requirements**

**RULE 4.3.15**

A VISA implementation **SHALL** support the access mode of opening a session with VI\_EXCLUSIVE\_LOCK.

**RULE 4.3.16**

**IF** a VISA implementation does not provide an external configuration utility to specify the attribute values **AND** viOpen() is invoked with the accessMode value set to VI\_LOAD\_CONFIG, **AND** the operation is successful, **THEN** the operation **SHALL** return VI\_WARN\_CONFIG\_NLOADED.

**OBSERVATION 4.3.6**

The VI\_LOAD\_CONFIG value provides a way to create a session with attribute values initialized other than the default values. An optional, external configuration utility is required to support this option.

**RULE 4.3.17**

A VISA implementation of viOpen() **SHALL** use a case-insensitive compare function when matching resource names against the name specified in rsrcName.

**OBSERVATION 4.3.7**

Calling viOpen() with "VXI::1::INSTR" will open the same resource as invoking it with "vxi::1::instr".

**RULE 4.3.18**

**IF** the accessMode parameter includes the flag VI\_EXCLUSIVE\_LOCK, a VISA implementation **SHALL** use the specified timeout parameter when acquiring the lock.

**PERMISSION 4.3.2**

A VISA implementation **MAY** use the timeout parameter when opening the resource, regardless of whether the VI\_EXCLUSIVE\_LOCK flag is specified.

**RECOMMENDATION 4.3.3**

If the value of the timeout parameter to viOpen is 0 and a VISA implementation uses the timeout when opening the resource, the implementation shouldbehave as if the timeout parameter is the VISA default timeout value of 2000 milliseconds.

**OBSERVATION 4.3.8**

It is optional to use the timeout parameter when opening network resources.

4.3.3.3 viParseRsrc(sesn, rsrcName, intfType, intfNum)

**Purpose**

Parse a resource string to get the interface information.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| sesn | IN | ViSession | Resource Manager session (should always be the Default Resource Manager for VISA returned from viOpenDefaultRM()). |
| rsrcName | IN | ViRsrc | Unique symbolic name of a resource. |
| intfType | OUT | ViUInt16 | Interface type of the given resource string. |
| intfNum | OUT | ViUInt16 | Board number of the interface of the given resource string. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Resource string is valid. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given sesn does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session. |
| VI\_ERROR\_INV\_RSRC\_NAME | Invalid resource reference specified. Parsing error. |
| VI\_ERROR\_RSRC\_NFOUND | Insufficient location information or resource not present in the system. |
| VI\_ERROR\_ALLOC | Insufficient system resources to parse the string. |
| VI\_ERROR\_LIBRARY\_NFOUND | A code library required by VISA could not be located or loaded. |
| VI\_ERROR\_INTF\_NUM\_NCONFIG | The interface type is valid but the specified interface number is not configured. |

**Description**

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

The values returned in intfType and intfNum correspond to the attributes VI\_ATTR\_INTF\_TYPE and VI\_ATTR\_INTF\_NUM. These values would be the same if a user opened that resource with viOpen() and queried the attributes with viGetAttribute().

**Related Items**

See also viFindRsrc(), viOpen(), and viParseRsrcEx().

**Implementation Requirements**

**RULE 4.3.19**

**IF** a VISA implementation recognizes aliases in viOpen(), **THEN** it **SHALL** recognize those same aliases in viParseRsrc().

**RECOMMENDATION 4.3.4**  
A VISA implementation should not perform any I/O to the specified resource during this operation. The recommended implementation of viParseRsrc() will return information determined solely from the resource string and any static configuration information (*e.g.*, .INI files or the Registry).

**RULE 4.3.20**

A VISA implementation of viParseRsrc() **SHALL** use a case-insensitive compare function when matching resource names against the name specified in rsrcName.

**OBSERVATION 4.3.9**

Calling viParseRsrc() with "VXI::1::INSTR" will produce the same results as invoking it with "vxi::1::instr".

4.3.3.4 viParseRsrcEx(sesn, rsrcName, intfType, intfNum, rsrcClass, unaliasedExpandedRsrcName, aliasIfExists)

**Purpose**

Parse a resource string to get extended interface information.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| sesn | IN | ViSession | Resource Manager session (should always be the Default Resource Manager for VISA returned from viOpenDefaultRM()). |
| rsrcName | IN | ViRsrc | Unique symbolic name of a resource. |
| intfType | OUT | ViUInt16 | Interface type of the given resource string. |
| intfNum | OUT | ViUInt16 | Board number of the interface of the given resource string. |
| rsrcClass | OUT | ViString | Specifies the resource class (for example, “INSTR”) of the given resource string, as defined in Section 5. |
| Unaliased Expanded RsrcName | OUT | ViString | This is the expanded version of the given resource string. The format should be similar to the VISA-defined canonical resource name. |
| aliasIf Exists | OUT | ViString | Specifies the user-defined alias for the given resource string, if a VISA implementation allows aliases and an alias exists for the given resource string. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Resource string is valid. |
| VI\_WARN\_EXT\_FUNC\_NIMPL | The operation succeeded, but a lower level driver did not implement the extended functionality. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given sesn does not support this operation. For VISA, this operation is supported only by the Default Resource Manager session. |
| VI\_ERROR\_INV\_RSRC\_NAME | Invalid resource reference specified. Parsing error. |
| VI\_ERROR\_RSRC\_NFOUND | Insufficient location information or resource not present in the system. |
| VI\_ERROR\_ALLOC | Insufficient system resources to parse the string. |
| VI\_ERROR\_LIBRARY\_NFOUND | A code library required by VISA could not be located or loaded. |
| VI\_ERROR\_INTF\_NUM\_NCONFIG | The interface type is valid but the specified interface number is not configured. |

**Description**

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

The values returned in intfType, intfNum, and rsrcClass correspond to the attributes VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_NUM, and VI\_ATTR\_RSRC\_CLASS. These values would be the same if a user opened that resource with viOpen() and queried the attributes with viGetAttribute().

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

The value returned in aliasIfExists allows programmatic access to user-defined aliases. If a VISA implementation does not implement aliases, the return value must be an empty string. If a VISA implementation allows multiple aliases for a single resource, then the implementation must pick one alias (in an implementation-defined manner) and return it in this parameter.

Table 4.3.3 Special Values for rsrcClass Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the resource class. |

Table 4.3.4 Special Values for unaliasedExpandedRsrcName Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the full resource name. |

Table 4.3.5 Special Values for aliasIfExists Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the alias. |

**Related Items**

See also viFindRsrc(), viOpen(), and viParseRsrc().

**Implementation Requirements**

**RULE 4.3.21**

**IF** a VISA implementation recognizes aliases in viOpen(), **THEN** it **SHALL** recognize those same aliases in viParseRsrcEx().

**RECOMMENDATION 4.3.5**  
A VISA implementation should not perform any I/O to the specified resource during this operation. The recommended implementation of viParseRsrcEx() will return information determined solely from the resource string and any static configuration information (*e.g.*, .INI files or the Registry).

**RULE 4.3.22**

A VISA implementation of viParseRsrcEx() **SHALL** use a case-insensitive compare function when matching resource names against the name specified in rsrcName.

**OBSERVATION 4.3.10**

Calling viParseRsrcEx() with "VXI::1::INSTR" will produce the same results as invoking it with "vxi::1::instr".

**OBSERVATION 4.3.11**

Calling viParseRsrc() with "VXI::BACKPLANE" may result in unaliasedExpandedRsrcName containing either "VXI0::BACKPLANE" or "VXI0::0::BACKPLANE". This is because the driver may not know the mainframe number until the resource is actually opened.

**RULE 4.3.23**

**IF** a VISA implementation of viParseRsrcEx() does not support aliases, **AND** the aliasIfExists parameter is not NULL, **THEN** the output value of aliasIfExists **SHALL** be an empty string.

**RULE 4.3.24**

**IF** a VISA implementation of viParseRsrcEx() supports multiple aliases per resource string, **AND** multiple aliases exist for the given rsrcName, **AND** the aliasIfExists parameter is not NULL, **THEN** the VISA implementation **SHALL** use one alias as the output value of aliasIfExists.

**RECOMMENDATION 4.3.6**

A VISA implementation should not allow the colon character (“:”) in user-defined aliases.

**PERMISSION 4.3.3**

A VISA implementation **MAY** allow the colon character (“:”) in user-defined aliases.

**OBSERVATION 4.3.12**

The intent of disallowing colons in aliases is that the VISA specification reserves that character for definition of all future canonical resource names. If a VISA implementation allows the user to enter a name that could later be defined as an actual resource name, then the behavior of such an alias could change in a way that users might not expect.

**OBSERVATION 4.3.13**

There are valid scenarios where a VISA implementation may want to allow colons in aliases. One such scenario is allowing one resource name to intentionally masquerade as another. However, an implementation that allows such behavior should take care to avoid user confusion over which resource is actually accessed when such an alias is defined.

**RULE 4.3.25**

The function viParseRsrcEx **SHALL** return unaliasedExpandedRsrcName in the format specified in this document.

**RULE 4.3.26**

A VISA implementation **SHALL** return PXI INSTR resource strings from viParseRsrc that include the function number, regardless of whether the PXI instrument has one or multiple functions.

**RULE 4.3.27**

A VISA implementation **SHALL** return USB INSTR resource strings from viParseRsrc that include the interface number, regardless of whether the USB instrument has one or multiple interfaces.

4.4 Search Services

VISA provides the ability to search and locate resources regardless of where the resource is residing. To be able to locate a resource in a VISA system, it is essential to be able to uniquely identify the given resource throughout the system. As described in Section 4.3, *Access Services*, a resource string is used for uniquely identifying a given resource in the system. In order to specify different variations of the resource strings to search for, the VISA Resource Manager allows the use of a regular expression to describe them.

4.4.1 Resource Regular Expression

A regular expression is a string consisting of ordinary characters as well as special characters. A regular expression is used for specifying patterns to match in a given string. Given a string and a regular expression, one can determine if the string matches the regular expression. A regular expression can also be used as a search criterion. Given a regular expression and a list of strings, one can match the regular expression against each string and return a list of strings that match the regular expression.

Tables 4.4.1 and 4.4.2 define the special characters and literals used in the grammar rules defined in this section and other sections of this document.

Table 4.4.1 Special Characters

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Symbol** |
| NL / LF | New Line / Line Feed | "\n" |
| HT | Horizontal Tab | "\t" |
| CR | Carriage Return | "\r" |
| FF | Form Feed | "\f" |
| SP | Blank Space | " " |

**OBSERVATION 4.4.1**

The definitions of character constants do not require any specific implementation. The implementor should follow language or industry standards as appropriate.

Table 4.4.2 Literals

|  |  |
| --- | --- |
| **Literal** | **Definition** |
| white\_space | NL, LF, HT, CR, FF, SP |
| digit | "0","1".."9" |
| letter | "a","b".."z", "A","B".."Z" |
| hex\_digit | "0","1".."9", "a","b".."f", "A","B".."F" |
| underscore | "\_" |

Table 4.4.3 Regular Expression Characters and Operators

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

**RULE 4.4.1**

The grouping operator () in a regular expression **SHALL** have the highest precedence.

**RULE 4.4.2**

The + and \* operators in a regular expression **SHALL** have the next highest precedence after the grouping operator.

**RULE 4.4.3**

The or operator | in a regular expression **SHALL** have the lowest precedence.

Table 4.4.4 Examples

|  |  |
| --- | --- |
| **Regular Expression** | **Sample Matches** |
| GPIB?\*INSTR | Matches GPIB0::2::INSTR, GPIB1::1::1::INSTR, and  GPIB-VXI1::8::INSTR. |
| GPIB[0-9]\*::?\*INSTR | Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not  GPIB-VXI1::8::INSTR. |
| GPIB[0-9]::?\*INSTR | Matches GPIB0::2::INSTR and GPIB1::1::1::INSTR but not GPIB12::8::INSTR. |

Table 4.4.4 Examples (continued)

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

**OBSERVATION 4.4.2**

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

4.4.2 Search Operations

viFindRsrc(sesn, expr, findList, retcnt, instrDesc)

viFindNext(findList, instrDesc)

**OBSERVATION 4.4.3**

For VISA, the local controller for VXI and GPIB-VXI interfaces will appear in the list of resources to find. The main purpose of this is to be able to access any shared memory that the controller exports as a VXI resource.

**OBSERVATION 4.4.4**

The non-immediate servants will also appear in the list of devices to find. For these devices, the attribute VI\_ATTR\_IMMEDIATE\_SERV will be set to VI\_FALSE.

4.4.2.1 viFindRsrc(sesn, expr, findList, retcnt, instrDesc)

**Purpose**

Query a VISA system to locate the resources associated with a specified interface*.*

**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 is a regular expression followed by an optional logical expression. The grammar for this expression is given below. |
| 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. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Resource(s) found. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given sesn does not support this operation. |
| VI\_ERROR\_INV\_EXPR | Invalid expression specified for search. |
| VI\_ERROR\_RSRC\_NFOUND | Specified expression does not match any devices. |

**Description**

This operation matches the value specified in the expr parameter with the resources available for a particular interface. On successful completion, it returns the first resource found in the list and returns a count to indicate if there were more resources found for the designated interface. This function also returns a handle to a find list. This handle points to the list of resources and it must be used as an input to viFindNext(). When this handle is no longer needed, it should be passed to viClose().

Table 4.4.5 Special Values for findList Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return a find list handle. |

Table 4.4.6 Special Values for retcnt Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of matches. |

The search criteria specified in the expr parameter has two parts: a regular expressionover a resource string (which is explained later), and an optional logical expression over attribute values. The regular expression is matched against the resource strings of resources known to the VISA Resource Manager. If the resource string matches the regular expression, the attribute values of the resource are then matched against the expression over attribute values. If the match is successful, the resource has met the search criteria and gets added to the list of resources found.

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

The syntax of expr is defined as follows:

Table 4.4.7 Special Characters and their Meaning

|  |  |
| --- | --- |
| **Special Character** | **Meaning** |
| && | Logical AND |
| || | Logical OR |
| ! | Logical negation (NOT) |
| () | Parenthesis |

expr :=

regularExpr ['{' attrExpr '}']

attrExpr :=

attrTerm |

attrExpr '||' attrTerm

attrTerm :=

attrFactor |

attrTerm '&&' attrFactor

attrFactor :=

'(' attrExpr ')' |

'!' attrFactor |

relationExpr

relationExpr :=

attributeId compareOp numValue |

attributeId equalityOp stringValue

compareOp :=

'==' | '!=' | '>' | '<' | '>=' | '<='

equalityOp :=

'==' | '!='

attributeId :=

character (character|digit|underscore)\*

numValue :=

digit+ |

'-' digit+ |

'0x' hex\_digit+ |

'0X' hex\_digit+

stringValue :=

'"' character\* '"'

regularExpr is defined in Section 4.4.1, *Resource Regular Expressions*.

**RULE 4.4.4**

The grouping operator () in a logical expression **SHALL** have the highest precedence.

**RULE 4.4.5**

The not operator ! in a logical expression **SHALL** have the next highest precedence after the grouping operator.

**RULE 4.4.6**

The or operator || in a logical expression **SHALL** have the lowest precedence.

Table 4.4.8 Examples

|  |  |
| --- | --- |
| **Expr** | **Meaning** |
| GPIB[0‑9]\*::?\*::?\*::INSTR {VI\_ATTR\_GPIB\_SECONDARY\_ADDR > 0} | Find all GPIB devices that have secondary addresses greater than 0. |
| ASRL?\*INSTR{VI\_ATTR\_ASRL\_BAUD == 9600} | Find all serial ports configured at 9600 baud. |
| ?\*VXI?\*INSTR{VI\_ATTR\_MANF\_ID == 0xFF6 && !(VI\_ATTR\_VXI\_LA == 0 || VI\_ATTR\_SLOT <= 0)} | Find all VXI instrument resources whose manufacturer ID is FF6 and who are not logical address 0, slot 0, or external controllers. |

**Related Items**

See viFindNext().

**Implementation Requirements**

**RULE 4.4.7**

Local attributes **SHALL** **NOT** be allowed in the logical expression part of the expr parameter to the viFindRsrc() operation.

**RULE 4.4.8**

**IF** the value VI\_NULL is specified in the findList parameter of viFindRsrc(), **AND** the return value is successful, **THEN** the VISA system **SHALL** automatically invoke viClose() on the find list handle rather than returning it to the application.

**OBSERVATION 4.4.5**

The findList and retCnt parameters to the viFindRsrc() operation are optional. This can be used if only the first match is important, and the number of matches is not needed.

**RULE 4.4.9**

A VISA implementation of viFindRsrc() **SHALL** use a case-insensitive compare function when matching resource names against the regular expression specified in expr.

**OBSERVATION 4.4.6**

Calling viFindRsrc() with "VXI?\*INSTR" will return the same resources as invoking it with "vxi?\*instr".

**PERMISSION 4.4.1**

A given implementation of viFindRsrc **MAY** return strings in formats other than those defined in this specification.

**OBSERVATION 4.4.7**

There are many ways that a vendor may want to return strings from viFindRsrc in an alternate format. One example is if the vendor has a configuration option to return aliases instead of canonical names. Another example is if the vendor chooses to omit optional portions of the resource name.

**OBSERVATION 4.4.8**

All resource strings returned by viFindRsrc() must be recognized by viParseRsrc() and viParseRsrcEx() and viOpen(). However, not all resource strings that can be parsed or opened have to be findable. Within these guidelines, it is acceptable for the exact behavior of viFindRsrc() to be modifiable through an optional, external configuration utility. For example, it is implementation dependent which (if any) VISA TCPIP resources a given implementation will return from viFindRsrc().

**RULE 4.4.10**

A VISA implementation that supports PXI INSTR resources **SHALL** match and return only one resource string per PXI INSTR resource.

**RULE 4.4.11**

VISA implementation that supports PXI INSTR **SHALL** be capable of returning the bus/device/function format for the string.

**PERMISSION 4.4.2**

A VISA implementation that supports PXI INSTR **MAY** provide configuration options to return other resource string formats for PXI resources, not limited to those defined in this specification, as long as only one resource string is returned per PXI resource.

4.4.2.2 viFindNext(findList, instrDesc)

**Purpose**

Return the next resource found during a previous call to viFindRsrc().

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| findList | IN | ViFindList | Describes a find list. This parameter must be created by viFindRsrc(). |
| 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. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Resource(s) found. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given findList does not support this operation. |
| VI\_ERROR\_RSRC\_NFOUND | There are no more matches. |

**Description**

This operation returns the next device found in the list created by viFindRsrc(). The list is referenced by the handle that was returned by viFindRsrc().

**Related Items**

See viFindRsrc().

**Implementation Requirements**

**RULE 4.4.12**

The findList passed to viFindNext() **SHALL** have been returned by viFindRsrc().

Section 5 VISA Resource Classes

The following sections define various resource classes that a complete VISA system, fully compliant with this specification, should implement. Since not all VISA implementations may implement all resource classes for all interfaces, the following rules and recommendations specify which classes are required for which interfaces.

# RULE 5.0.1

**IF** a VISA implementation supports the GPIB interface (VI\_INTF\_GPIB), **THEN** it **SHALL** implement the resource types INSTR and INTFC.

**RECOMMENDATION 5.0.1**

If a VISA implementation supports the GPIB interface (VI\_INTF\_GPIB), it should also implement the resource type SERVANT.

**RULE 5.0.2**

**IF** a VISA implementation supports the VXI interface (VI\_INTF\_VXI), **THEN** it **SHALL** implement the resource types INSTR and MEMACC.

**RECOMMENDATION 5.0.2**

If a VISA implementation supports the VXI interface (VI\_INTF\_VXI), it should also implement the resource types BACKPLANE and SERVANT.

**RULE 5.0.3**

**IF** a VISA implementation supports the GPIB-VXI interface (VI\_INTF\_GPIB\_VXI), **THEN** it **SHALL** implement the resource types INSTR and MEMACC.

**RECOMMENDATION 5.0.3**

If a VISA implementation supports the GPIB-VXI interface (VI\_INTF\_GPIB\_VXI), it should also implement the resource type BACKPLANE.

**RULE 5.0.4**

**IF** a VISA implementation supports the Serial interface (VI\_INTF\_ASRL), **THEN** it **SHALL** implement the resource type INSTR.

**RULE 5.0.5**

**IF** a VISA implementation supports the TCPIP interface (VI\_INTF\_TCPIP), **THEN** it **SHALL** implement the resource types INSTR and SOCKET.

**RECOMMENDATION 5.0.4**

If a VISA implementation supports the TCPIP interface (VI\_INTF\_TCPIP), it should also implement the resource type SERVANT.

**RULE 5.0.6**

**IF** a VISA implementation supports the USB interface (VI\_INTF\_USB), **THEN** it **SHALL** implement the resource type INSTR.

**RULE 5.0.7**

**IF** a VISA implementation supports the PXI interface (VI\_INTF\_PXI), **THEN** it **SHALL** implement the resource types INSTR and MEMACC.

**RECOMMENDATION 5.0.5**

If a VISA implementation supports the PXI interface (VI\_INTF\_PXI), it should also implement the resource type BACKPLANE.

5.1 Instrument Control Resource

This section describes the resource that is provided to encapsulate the various operations of a device (reading, writing, triggering, and so on). A VISA Instrument Control (INSTR) Resource, like any other resource, defines the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task, such as sending a string to a message-based device.

5.1.1 INSTR Resource Overview

The INSTR Resource lets a controller interact with the device associated with this resource, by providing the controller with services to send blocks of data to the device, request blocks of data from the device, send the device clear command to the device, trigger the device, and find information about the device’s status. In addition, it allows the controller to access registers on devices that reside on memory-mapped buses. These services are described in detail in the remainder of this section.

• **Basic I/O Services**

– The Read Service lets a controller request blocks of data from the device that is associated with this resource. How the returned data is interpreted depends on how the device has been programmed—for example, messages, commands, or binary encoded data. The resource receives data in the native mode of the interface it is associated with. It also permits implementations that provide alternate modes supported by the interface. Setting the appropriate attribute modifies the data transmittal method and other features, such as setting the termination character.

– The Write Service lets a controller send blocks of data to the device associated with this resource. The device can interpret the data as necessary—for example, messages, commands, or binary encoded data. The resource sends data in the native mode of the interface it is associated with. It also permits implementations that provide alternate modes supported by the interface. Setting the appropriate attribute modifies the data transmittal method and other features, such as specifying whether to send an END indicator with each block of data.

– The Trigger Service provides monitoring and control access to the trigger capabilities of the device associated with the resource. Assertion of both software and hardware triggers is handled by using the viAssertTrigger() operation. (See the operation listing for more information.)

– The Status/Service Request Service allows the controller to service requests made by the other service requesters in a system. In this role of a service provider, it can procure the device status information. Applications can use the viReadSTB() operation to manually obtain the status information. If the resource cannot obtain the status information from the requester in the actual timeout period, timeout is returned.

– The Clear Service lets a controller send the device clear command to the device it is associated with, as specified by the interface regulations and the type of device. For a GPIB device, this amounts to sending the IEEE 488.1 *SDC* (04h) command; for a VXI or MXI device, it amounts to sending the Word Serial command *Clear* (FFFFh)*.* The action that the device takes depends on the interface to which it is connected.

**• Formatted I/O Services**

– The Formatted I/O Services perform formatted and buffered I/O for devices. A formatted write operation writes to a buffer, while a formatted read operation reads from a buffer. Buffering improves system performance by making it possible to transfer large blocks of data to and from devices. The system provides separate read and write buffers that can be disabled or have their sizes modified by a user application, via the viSetBuf() operation.

The following section describes buffer maintenance and buffer flushing issues that are related to formatted I/O resources. The descriptions apply to all buffered read and buffered write operations. For example, the viPrintf() description applies equally to other buffered write operations (viVPrintf() and viBufWrite()). Similarly, the viScanf() description applies to other buffered read operations (viVScanf() and viBufRead()).

**RULE 5.1.1**

All formatted write operations (viPrintf(), viVPrintf(), and viBufWrite()) **SHALL** use the same write buffer for a corresponding session.

**RULE 5.1.2**

All formatted read operations (viScanf(), viVScanf(), and viBufRead()) **SHALL** use the same read buffer for a corresponding session.

**RULE 5.1.3**

The write buffer used in the formatted buffered write operations **SHALL** be unique per session.

**RULE 5.1.4**

The read buffer used in the formatted buffered read operations **SHALL** be unique per session.

**RULE 5.1.5**

The write buffer used in the buffered write operation **SHALL NOT** be same as the read buffer used in the read operations.

Although you can explicitly flush the buffers by making a call to viFlush(), the buffers are flushed implicitly under some conditions. These conditions vary for the viPrintf() and viScanf() operations.

Flushing a write buffer immediately sends any queued data to the device. The write buffer is maintained by the viPrintf() operation. To explicitly flush the write buffer, you can make a call to the viFlush() operation with a write flag set.

**RULE 5.1.6**

The write buffer **SHALL** be flushed automatically under the following conditions:

• When an END-indicator character is sent.

• When the buffer is full.

• In response to a call to viSetBuf() with the VI\_WRITE\_BUF flag set.

**RULE 5.1.7**

When the write buffer is flushed automatically because the buffer is full, the write buffer **SHALL** ensure there is more data to be sent later.

**OBSERVATION 5.1.1**

RULE 5.1.7 ensures that if the user calls viPrintf() and the buffer fills up, and then the user explicitly calls viFlush(), that the END indicator being sent with the explicit flush has data that it can go with. This is necessary because the 488.2 END indicator is not data all on its own.

Flushing a read buffer discards the data in the read buffer. This guarantees that the next call to a viScanf() (or related) operation reads data directly from the device rather than from queued data residing in the read buffer. The read buffer is maintained by the viScanf() operation. To explicitly flush the read buffer, you can make a call to the viFlush() operation with a read flag set.

The formatted I/O buffers of a session to a given device are reset whenever that device is cleared. At such a time, the read and write buffer must be flushed and any ongoing operation through the read/write port must be aborted.

**RULE 5.1.8**

An invocation of a viClear() operation on a resource **SHALL** flush the read buffer and discard the contents of the write buffers.

• **Memory I/O Services**

– The High-Level Access Service allows register-level access to devices on interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or even VXI or VME devices controlled by a GPIB-to-VXI device. A resource exists for each interface to which the controller has access. When dealing with memory accesses, there is a tradeoff between speed and complexity, and between software overhead and encapsulation. The High-Level Access Service is similar in purpose to the Low-Level Access Service. The difference between these two services is that the High-Level Access Service has greater software overhead because it encapsulates most of the code required to perform the memory access, such as window mapping and error checking. In general, high-level accesses are slower than low-level accesses, but they encapsulate the operations necessary to perform the access and are considered safer.

The High-Level Access Service lets the programmer access memory on the interface bus through simple operations such as viIn16() and viOut16(). These operations encapsulate the map/unmap and peek/poke operations found in the Low-Level Access Service. There is no need to explicitly map the memory to a window.

– The Low-Level Access Service, like the High-Level Access Service, allows register-level access to devices on interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or VME or VXI memory through a system controlled by a GPIB-to-VXI controller. A resource exists for each interface of this type that the controller has locally. When dealing with memory accesses, there is a tradeoff between speed and complexity and between software overhead and encapsulation. The Low-Level Access Service is similar in purpose to the High-Level Access Service. The difference between these two services is that the Low‑Level Access Service increases access speed by removing software overhead, but requires more programming effort by the user. To decrease the amount of overhead involved in the memory access, the Low-Level Access Service does not return any error information in the access operations.

Before an application can use the Low-Level Access Service on the interface bus, it must map a range of addresses using the operation viMapAddress(). Although the resource handles the allocation and operation of the window, the programmer must free the window via viUnmapAddress() when finished. This makes the window available for the system to reallocate.

**RULE 5.1.9**

**IF** an application performs viClose() on a session with memory still mapped, **THEN** viClose() **SHALL** perform an implicit unmapping of the mapped window.

**• Shared Memory Services**

– The Shared Memory Service allows users to allocate memory on a particular device to be used exclusively by that session. The viMemAlloc() operation allows such an allocation, by specifying the size. The space in which the memory is located is that which is exported by the device to a given bus. The viMemFree() operation allows the user to free memory previously allocated using viMemAlloc().

**RULE 5.1.10**

**IF** an application performs viClose() on a session with shared memory still allocated, **THEN** viClose() **SHALL** perform an implicit freeing up of the allocated region(s).

5.1.2 INSTR Resource Attributes

**Generic INSTR Resource Attributes**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | | ViUInt16 | VI\_INTF\_VXI  VI\_INTF\_GPIB  VI\_INTF\_GPIB\_VXI  VI\_INTF\_ASRL  VI\_INTF\_PXI  VI\_INTF\_TCPIP  VI\_INTF\_USB |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | | ViString | N/A |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_TRIG\_ID | R/W\* | Local | | ViInt16 | VI\_TRIG\_SW; VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_VXI0 to  VI\_TRIG\_STAR\_VXI2;  VI\_TRIG\_STAR\_INSTR |
| VI\_ATTR\_DMA\_ALLOW\_EN | RW | | Local | ViBoolean | VI\_TRUE  VI\_FALSE |

\* The attribute VI\_ATTR\_TRIG\_ID is R/W (readable and writeable) when the corresponding session is not enabled to receive trigger events. When the session is enabled to receive trigger events, the attribute VI\_ATTR\_TRIG\_ID is RO (read only).

**Message-Based INSTR Resource Attributes**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | | **Data Type** | **Range** |
| VI\_ATTR\_FILE\_APPEND\_EN | RW | | Local | ViBoolean | VI\_TRUE VI\_FALSE |
| VI\_ATTR\_IO\_PROT | R/W | Local | | ViUInt16 | VI\_PROT\_NORMAL  VI\_PROT\_FDC  VI\_PROT\_HS488  VI\_PROT\_4882\_STRS  VI\_PROT\_USBTMC\_VENDOR |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | R/W | Local | | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_DISABLE |
| VI\_ATTR\_RD\_BUF\_SIZE | RO | | Local | ViUInt32 | N/A |
| VI\_ATTR\_SEND\_END\_EN | R/W | Local | | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_SUPPRESS\_END\_EN | R/W | Local | | ViBoolean | VI\_TRUE  VI\_FALSE |

(continues)

**Message-Based INSTR Resource Attributes (Continued)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | | **Data Type** | **Range** |
| VI\_ATTR\_TERMCHAR | R/W | Local | | ViUInt8 | 0 to FFh |
| VI\_ATTR\_TERMCHAR\_EN | R/W | Local | | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | R/W | Local | | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_WR\_BUF\_SIZE | RO | | Local | ViUInt32 | N/A |

**GPIB and GPIB-VXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | RO | Global | ViUInt16 | 0 to 30 |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | RO | Global | ViUInt16 | 0 to 31, VI\_NO\_SEC\_ADDR |
| VI\_ATTR\_GPIB\_READDR\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_GPIB\_UNADDR\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_GPIB\_REN\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |

**VXI and GPIB-VXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_MAINFRAME\_LA | RO | Global | ViInt16 | 0 to 255  VI\_UNKNOWN\_LA |
| VI\_ATTR\_MEM\_Base\_32 | RO | Global | ViUInt32 | N/A |
| VI\_ATTR\_MEM\_Base\_64 | RO | Global | ViBusAddress64 | N/A |
| VI\_ATTR\_MEM\_Size\_32 | RO | Global | ViUInt32 | N/A |
| VI\_ATTR\_MEM\_Size\_64 | RO | Global | ViBusSize64 | N/A |
| VI\_ATTR\_MEM\_Space | RO | Global | ViUInt16 | VI\_A16\_SPACE  VI\_A24\_SPACE  VI\_A32\_SPACE  VI\_A64\_SPACE |
| VI\_ATTR\_VXI\_LA | RO | Global | ViInt16 | 0 to 511 |
| VI\_ATTR\_CMDR\_LA | RO | Global | ViInt16 | 0 to 255  VI\_UNKNOWN\_LA |
| VI\_ATTR\_IMMEDIATE\_SERV | RO | Global | viBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_FDC\_CHNL | R/W | Local | ViUInt16 | 0 to 7 |
| VI\_ATTR\_FDC\_GEN\_SIGNAL\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |

(continues)

**VXI and GPIB-VXI Specific INSTR Resource Attributes (Continued)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** | |
| VI\_ATTR\_FDC\_MODE | R/W | Local | ViUInt16 | VI\_FDC\_NORMAL  VI\_FDC\_STREAM |
| VI\_ATTR\_FDC\_USE\_PAIR | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_SRC\_BYTE\_ORDER | R/W | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |
| VI\_ATTR\_DEST\_BYTE\_ORDER | R/W | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |
| VI\_ATTR\_WIN\_BYTE\_ORDER | R/W\* | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |
| VI\_ATTR\_SRC\_ACCESS\_PRIV | R/W | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV  VI\_D64\_NPRIV  VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 |
| VI\_ATTR\_DEST\_ACCESS\_PRIV | R/W | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV  VI\_D64\_NPRIV  VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 |
| VI\_ATTR\_WIN\_ACCESS\_PRIV | R/W\* | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV |
| VI\_ATTR\_VXI\_DEV\_CLASS | RO | Global | ViUInt16 | VI\_VXI\_CLASS\_MEMORY VI\_VXI\_CLASS\_EXTENDED VI\_VXI\_CLASS\_MESSAGE VI\_VXI\_CLASS\_REGISTER VI\_VXI\_CLASS\_OTHER |
| VI\_ATTR\_VXI\_TRIG\_SUPPORT | RO | Global | ViUInt32 | N/A |

\* For VISA 2.2, the attributes VI\_ATTR\_WIN\_BYTE\_ORDER and VI\_ATTR\_WIN\_ACCESS\_PRIV are R/W (readable and writeable) when the corresponding session is not mapped (VI\_ATTR\_WIN\_ACCESS == VI\_NMAPPED). When the session is mapped, these attributes are RO (read only).

**GPIB-VXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_PARENT\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |

**ASRL Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_ASRL\_AVAIL\_NUM | RO | Global | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_ASRL\_BAUD | R/W | Global | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_ASRL\_DATA\_BITS | R/W | Global | ViUInt16 | 5 to 8 |
| VI\_ATTR\_ASRL\_PARITY | R/W | Global | ViUInt16 | VI\_ASRL\_PAR\_NONE  VI\_ASRL\_PAR\_ODD  VI\_ASRL\_PAR\_EVEN  VI\_ASRL\_PAR\_MARK  VI\_ASRL\_PAR\_SPACE |
| VI\_ATTR\_ASRL\_STOP\_BITS | R/W | Global | ViUInt16 | VI\_ASRL\_STOP\_ONE  VI\_ASRL\_STOP\_ONE5  VI\_ASRL\_STOP\_TWO |
| VI\_ATTR\_ASRL\_FLOW\_CNTRL | R/W | Global | ViUInt16 | VI\_ASRL\_FLOW\_NONE  VI\_ASRL\_FLOW\_XON\_XOFF  VI\_ASRL\_FLOW\_RTS\_CTS  VI\_ASRL\_FLOW\_DTR\_DSR |
| VI\_ATTR\_ASRL\_END\_IN | R/W | Local | ViUInt16 | VI\_ASRL\_END\_NONE  VI\_ASRL\_END\_LAST\_BIT  VI\_ASRL\_END\_TERMCHAR |
| VI\_ATTR\_ASRL\_END\_OUT | R/W | Local | ViUInt16 | VI\_ASRL\_END\_NONE  VI\_ASRL\_END\_LAST\_BIT  VI\_ASRL\_END\_TERMCHAR  VI\_ASRL\_END\_BREAK |
| VI\_ATTR\_ASRL\_CTS\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_ASRL\_DCD\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_ASRL\_DSR\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_ASRL\_DTR\_STATE | R/W | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_ASRL\_RI\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_ASRL\_RTS\_STATE | R/W | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |

(continues)

**ARSL Specific INSTR Resource Attributes (Continued)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_ASRL\_REPLACE\_CHAR | R/W | Local | ViUInt8 | 0 to FFh |
| VI\_ATTR\_ASRL\_XON\_CHAR | R/W | Local | ViUInt8 | 0 to FFh |
| VI\_ATTR\_ASRL\_XOFF\_CHAR | R/W | Local | ViUInt8 | 0 to FFh |

**TCPIP Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_TCPIP\_ADDR | RO | Global | ViString | N/A |
| VI\_ATTR\_TCPIP\_HOSTNAME | RO | Global | ViString | N/A |
| VI\_ATTR\_TCPIP\_DEVICE\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_TCPIP\_IS\_HISLIP | RO | Global | ViBoolean | VI\_TRUE, VI\_FALSE |

**VXI and GPIB-VXI and USB Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_4882\_COMPLIANT | RO | Global | ViBoolean | VI\_TRUE  VI\_FALSE |

**VXI and GPIB-VXI and USB and PXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_MANF\_ID | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_MODEL\_CODE | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_MANF\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_MODEL\_NAME | RO | Global | ViString | N/A |

**USB Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_USB\_SERIAL\_NUM | RO | Global | ViString | N/A |
| VI\_ATTR\_USB\_INTFC\_NUM | RO | Global | ViInt16 | 0 to 254 |
| VI\_ATTR\_USB\_MAX\_INTR\_SIZE | RW | Local | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_USB\_PROTOCOL | RO | Global | ViInt16 | 0 to 255 |

**VXI and GPIB-VXI and PXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_Slot | RO | Global | ViInt16 | 0 to 18  VI\_UNKNOWN\_SLOT |
| VI\_ATTR\_SRC\_INCREMENT | R/W | Local | ViInt32 | 0 to 1 |
| VI\_ATTR\_DEST\_INCREMENT | R/W | Local | ViInt32 | 0 to 1 |
| VI\_ATTR\_WIN\_ACCESS | RO | Local | ViUInt16 | VI\_NMAPPED  VI\_USE\_OPERS  VI\_DEREF\_ADDR |
| VI\_ATTR\_WIN\_BASE\_ADDR\_32 | RO | Local | ViBusAddress | N/A |
| VI\_ATTR\_WIN\_BASE\_ADDR\_64 | RO | Local | ViBusAddress64 | N/A |
| VI\_ATTR\_WIN\_SIZE\_32 | RO | Local | ViBusSize | N/A |
| VI\_ATTR\_WIN\_SIZE\_64 | RO | Local | ViBusSize64 | N/A |

**PXI Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbolic Name | Access Privilege | | Data Type | Range |
| VI\_ATTR\_PXI\_BUS\_NUM | RO | Global | ViUInt16 | 0 to 255 |
| VI\_ATTR\_PXI\_DEV\_NUM | RO | Global | ViUInt16 | 0 to 31 |
| VI\_ATTR\_PXI\_FUNC\_NUM | RO | Global | ViUInt16 | 0 to 7 |
| VI\_ATTR\_PXI\_SLOTPATH | RO | Global | ViString | N/A |
| VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_SLOT |
| VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_SLOT |
| VI\_ATTR\_PXI\_TRIG\_BUS | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_TRIG |
| VI\_ATTR\_PXI\_STAR\_TRIG\_BUS | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_TRIG |
| VI\_ATTR\_PXI\_STAR\_TRIG\_LINE | RO | Global | ViInt16 | 0 to 32767  VI\_UNKNOWN\_TRIG |
| VI\_ATTR\_PXI\_MEM\_TYPE\_BAR*n* (where *n* is 0,1,2,3,4,5) | RO | Global | ViUInt16 | VI\_PXI\_ADDR\_MEM, VI\_PXI\_ADDR\_IO, VI\_PXI\_ADDR\_NONE |

(continues)

**PXI Specific INSTR Resource Attributes (Continued)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbolic Name | Access Privilege | | Data Type | Range |
| VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n* (where *n* is 0,1,2,3,4,5) | RO | Global | ViBusAddress | N/A |
| VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_32 (where *n* is 0,1,2,3,4,5) | RO | Global | ViUInt32 | N/A |
| VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_64 (where *n* is 0,1,2,3,4,5) | RO | Global | ViBusAddress64 | N/A |
| VI\_ATTR\_PXI\_MEM\_SIZE\_BAR*n*\_32 (where *n* is 0,1,2,3,4,5) | RO | Global | ViUInt32 | N/A |
| VI\_ATTR\_PXI\_MEM\_SIZE\_BAR*n* \_64 (where *n* is 0,1,2,3,4,5) | RO | Global | ViBusSize64 | N/A |
| VI\_ATTR\_PXI\_CHASSIS | RO | Global | ViInt16 | 1 to 32767 VI\_UNKNOWN\_CHASSIS |
| VI\_ATTR\_PXI\_IS\_EXPRESS | RO | Global | ViBoolean | VI\_TRUE, VI\_FALSE |
| VI\_ATTR\_PXI\_SLOT\_LWIDTH | RO | Global | ViInt16 | 1, 4, 8 |
| VI\_ATTR\_PXI\_MAX\_LWIDTH | RO | Global | ViInt16 | 1, 4, 8 |
| VI\_ATTR\_PXI\_ACTUAL\_LWIDTH | RO | Global | ViInt16 | 1, 4, 8 |
| VI\_ATTR\_PXI\_DSTAR\_BUS | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_TRIG |
| VI\_ATTR\_PXI\_DSTAR\_SET | RO | Global | ViInt16 | 0 to 32767 VI\_UNKNOWN\_TRIG |
| VI\_ATTR\_PXI\_ALLOW\_WRITE\_COMBINe | RW | Local | ViBoolean | VI\_TRUE, VI\_FALSE |

**HiSLIP Specific INSTR Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_TCPIP\_HISLIP\_OVERLAP\_EN | R/W | Local | ViBoolean | VI\_TRUE, VI\_FALSE |
| VI\_ATTR\_TCPIP\_HISLIP\_VERSION | RO | Local | ViVersion | N/A |
| VI\_ATTR\_TCPIP\_HISLIP\_MAX\_MESSAGE\_KB | R/W | Local | ViUInt32 | 0h – ffffffffh |

**Attribute Descriptions**

**Generic INSTR Resource Attributes**

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_TRIG\_ID Identifier for the current triggering mechanism.

VI\_ATTR\_DMA\_ALLOW\_EN This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

**Message-Based INSTR Resource Attributes**

VI\_ATTR\_FILE\_APPEND\_EN This attribute specifies whether viReadToFile() will overwrite (truncate) or append when opening a file.

VI\_ATTR\_IO\_PROT Specifies which protocol to use. In VXI systems, for example, you can choose between normal word serial or fast data channel (FDC). In GPIB, you can choose between normal and high‑speed (HS488) data transfers. In ASRL and TCPIP systems, you can choose between normal and 488-style transfers, in which case the viAssertTrigger() and viReadSTB() operations send 488.2-defined strings.

VI\_ATTR\_RD\_BUF\_OPER\_MODE Determines the operational mode of the read buffer. When the operational mode is set to VI\_FLUSH\_DISABLE (default), the buffer is flushed only on explicit calls to viFlush().

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the buffer is flushed every time a viScanf() operation completes.

VI\_ATTR\_RD\_BUF\_SIZE This attribute specifies the size of the formatted I/O read buffer. The user can modify this value by calling viSetBuf().

VI\_ATTR\_SEND\_END\_EN Whether to assert END during the transfer of the last byte of the buffer.

VI\_ATTR\_SUPPRESS\_END\_EN Whether to suppress the END indicator termination. If this attribute is set to VI\_TRUE, the END indicator does not terminate read operations. If this attribute is set to VI\_FALSE, the END indicator terminates read operations.

VI\_ATTR\_TERMCHAR Termination character. When the termination character is read and VI\_ATTR\_TERMCHAR\_EN is enabled during a read operation, the read operation terminates.

VI\_ATTR\_TERMCHAR\_EN Flag that determines whether the read operation should terminate when a termination character is received.

VI\_ATTR\_WR\_BUF\_OPER\_MODE Determines the operational mode of the write buffer. When the operational mode is set to VI\_FLUSH\_WHEN\_FULL (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up.

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the write buffer is flushed under the same conditions, and also every time a viPrintf() operation completes.

VI\_ATTR\_WR\_BUF\_SIZE This attribute specifies the size of the formatted I/O write buffer. The user can modify this value by calling viSetBuf().

**GPIB and GPIB-VXI Specific INSTR Resource Attributes**

VI\_ATTR\_GPIB\_PRIMARY\_ADDR Primary address of the GPIB device used by the given session.

VI\_ATTR\_GPIB\_SECONDARY\_ADDR Secondary address of the GPIB device used by the given session.

VI\_ATTR\_GPIB\_READDR\_EN This attribute specifies whether to use repeat addressing before each read or write operation.

VI\_ATTR\_GPIB\_UNADDR\_EN This attribute specifies whether to unaddress the device (UNT and UNL) after each read or write operation.

VI\_ATTR\_GPIB\_REN\_STATE This attribute returns the current state of the GPIB REN interface line.

**VXI and GPIB-VXI Specific INSTR Resource Attributes**

VI\_ATTR\_MAINFRAME\_LA This is the logical address of a given device in the mainframe, usually the device with the lowest logical address. Other possible values include the logical address of the slot-0 controller or of the parent-side extender. Often, these are all the same value. The purpose of this attribute is to provide a unique ID for each mainframe. A VISA manufacturer can choose any of these values, but must be consistent across mainframes. If this value is not known, the attribute value returned is VI\_UNKNOWN\_LA.

VI\_ATTR\_MEM\_Base\_64

VI\_ATTR\_MEM\_Base\_32 Base address of the device in VXIbus memory address space. This base address is applicable to A24 or A32 or A64 address space.

VI\_ATTR\_MEM\_Size\_64

VI\_ATTR\_MEM\_Size\_32 Size of memory requested by the device in VXIbus address space.

VI\_ATTR\_MEM\_Space VXIbus address space used by the device. The four types are A16 only, A16/A24, A16/A32, or A16/A64 memory address space.

VI\_ATTR\_VXI\_LA Logical address of the VXI or VME device used by the given session. For a VME device, the logical address is actually a pseudo-address in the range 256 to 511.

VI\_ATTR\_CMDR\_LA Logical address of the commander of the VXI device used by the given session.

VI\_ATTR\_IMMEDIATE\_SERV Specifies whether the given device is an immediate servant of the controller running VISA.

VI\_ATTR\_FDC\_CHNL This attribute determines which FDC channel will be used to transfer the buffer.

VI\_ATTR\_FDC\_SIGNAL\_GEN\_EN Setting this attribute to VI\_TRUE lets the servant send a signal when control of the FDC channel is passed back to the commander. This action frees the commander from having to poll the FDC header while engaging in an FDC transfer.

VI\_ATTR\_FDC\_MODE This attribute determines which FDC mode to use (Normal mode or Stream mode).

VI\_ATTR\_FDC\_USE\_PAIR If set to VI\_TRUE, a channel pair will be used for transferring data. Otherwise, only one channel will be used.

VI\_ATTR\_SRC\_BYTE\_ORDER This attribute specifies the byte order to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.

VI\_ATTR\_DEST\_BYTE\_ORDER This attribute specifies the byte order to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.

VI\_ATTR\_WIN\_BYTE\_ORDER This attribute specifies the byte order to be used in low-level access operations, such as viMapAddress(), viPeek*XX*() and viPoke*XX*(), when accessing the mapped window.

VI\_ATTR\_SRC\_ACCESS\_PRIV This attribute specifies the address modifier to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.

VI\_ATTR\_DEST\_ACCESS\_PRIV This attribute specifies the address modifier to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.

VI\_ATTR\_WIN\_ACCESS\_PRIV This attribute specifies the address modifier to be used in low-level access operations, such as viMapAddress(), viPeek*XX*() and viPoke*XX*(), when accessing the mapped window.

VI\_ATTR\_VXI\_DEV\_CLASS This attribute represents the VXI-defined device class to which the resource belongs, either message based (VI\_VXI\_CLASS\_MESSAGE), register based (VI\_VXI\_CLASS\_REGISTER), extended (VI\_VXI\_CLASS\_EXTENDED), or memory (VI\_VXI\_CLASS\_MEMORY). VME devices are usually either register based or belong to a miscellaneous class (VI\_VXI\_CLASS\_OTHER).

VI\_ATTR\_VXI\_TRIG\_SUPPORT This attribute shows which VXI trigger lines this implementation supports. This is a bit vector. Bits 0-7 correspond to VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7. Bits 8-13 correspond to VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5. Bits 14-25 correspond to VI\_TRIG\_STAR\_SLOT1 to VI\_TRIG\_STAR\_SLOT12. Bit 27 corresponds to VI\_TRIG\_PANEL\_IN and bit 28 corresponds to VI\_TRIG\_PANEL\_OUT. Bits 29-31 correspond to VI\_TRIG\_STAR\_VXI0 to VI\_TRIG\_STAR\_VXI2.

**GPIB-VXI Specific INSTR Resource Attributes**

VI\_ATTR\_INTF\_PARENT\_NUM Board number of the GPIB board to which the GPIB-VXI is attached.

**ASRL Specific INSTR Resource Attributes**

VI\_ATTR\_ASRL\_AVAIL\_NUM This attribute shows the number of bytes available in the global receive buffer.

VI\_ATTR\_ASRL\_BAUD This is the baud rate of the interface. It is represented as an unsigned 32-bit integer so that any baud rate can be used, but it usually requires a commonly used rate such as 300, 1200, 2400, or 9600 baud.

VI\_ATTR\_ASRL\_DATA\_BITS This is the number of data bits contained in each frame (from 5 to 8). The data bits for each frame are located in the low‑order bits of every byte stored in memory.

VI\_ATTR\_ASRL\_PARITY This is the parity used with every frame transmitted and received. VI\_ASRL\_PAR\_MARK means that the parity bit exists and is always 1. VI\_ASRL\_PAR\_SPACE means that the parity bit exists and is always 0.

VI\_ATTR\_ASRL\_STOP\_BITS This is the number of stop bits used to indicate the end of a frame. The value VI\_ASRL\_STOP\_ONE5 indicates one-and-one-half (1.5) stop bits.

VI\_ATTR\_ASRL\_FLOW\_CNTRL If this attribute is set to VI\_ATTR\_ASRL\_FLOW\_NONE, the transfer mechanism does not use flow control, and buffers on both sides of the connection are assumed to be large enough to hold all data transferred.

If this attribute is set to VI\_ATTR\_ASRL\_FLOW\_XON\_XOFF, the transfer mechanism uses the XON and XOFF characters to perform flow control. The transfer mechanism controls input flow by sending XOFF when the receive buffer is nearly full, and it controls the output flow by suspending transmission when XOFF is received.

If this attribute is set to VI\_ATTR\_ASRL\_FLOW\_RTS\_CTS, the transfer mechanism uses the RTS output signal and the CTS input signal to perform flow control. The transfer mechanism controls input flow by unasserting the RTS signal when the receive buffer is nearly full, and it controls output flow by suspending the transmission when the CTS signal is unasserted.

If this attribute is set to VI\_ASRL\_FLOW\_DTR\_DSR, the transfer mechanism uses the DTR output signal and the DSR input signal to perform flow control. The transfer mechanism controls input flow by unasserting the DTR signal when the receive buffer is nearly full, and it controls output flow by suspending the transmission when the DSR signal is unasserted.

This attribute can specify multiple flow control mechanisms by bit-ORing multiple values together. However, certain combinations may not be supported by all serial ports and/or operating systems.

VI\_ATTR\_ASRL\_END\_IN This attribute indicates the method used to terminate read operations. If it is set to VI\_ASRL\_END\_NONE, the read will not terminate until all of the requested data is received (or an error occurs). If it is set to VI\_ASRL\_END\_TERMCHAR, the read will terminate as soon as the character in VI\_ATTR\_TERMCHAR is received. If it is set to VI\_ASRL\_END\_LAST\_BIT, the read will terminate as soon as a character arrives with its last bit set. For example, if VI\_ATTR\_ASRL\_DATA\_BITS is set to 8, then the read will terminate when a character arrives with the 8th bit set.

VI\_ATTR\_ASRL\_END\_OUT This attribute indicates the method used to terminate write operations. If it is set to VI\_ASRL\_END\_NONE, the write will not append anything to the data being written. If it is set to VI\_ASRL\_END\_BREAK, the write will transmit a break after all the characters for the write have been sent. If it is set to VI\_ASRL\_END\_LAST\_BIT, the write will send all but   
the last character with the last bit clear, then transmit   
the last character with the last bit set. For example, if VI\_ATTR\_ASRL\_DATA\_BITS is set to 8, then the write will clear the 8th bit for all but the last character, then transmit   
the last character with the 8th bit set. If it is set to VI\_ASRL\_END\_TERMCHAR, the write will send the character in VI\_ATTR\_TERMCHAR after the data being transmitted.

VI\_ATTR\_ASRL\_CTS\_STATE This attribute shows the current state of the Clear To Send (CTS) input signal.

VI\_ATTR\_ASRL\_RTS\_STATE This attribute is used to manually assert or unassert the Request To Send (RTS) output signal. When the VI\_ATTR\_ASRL\_FLOW\_CNTRL attribute is set to VI\_ASRL\_FLOW\_RTS\_CTS, this attribute is ignored when changed, but can be read to determine whether the background flow control is asserting or unasserting the signal.

VI\_ATTR\_ASRL\_DTR\_STATE This attribute is used to manually assert or unassert the Data Terminal Ready (DTR) output signal.

VI\_ATTR\_ASRL\_DSR\_STATE This attribute shows the current state of the Data Set Ready (DSR) input signal.

VI\_ATTR\_ASRL\_DCD\_STATE This attribute shows the current state of the Data Carrier Detect (DCD) input signal. The DCD signal is often used by modems to indicate the detection of a carrier (remote modem) on the telephone line. The DCD signal is also known as “Receive Line Signal Detect (RLSD).”

VI\_ATTR\_ASRL\_RI\_STATE This attribute shows the current state of the Ring Indicator (RI) input signal. The RI signal is often used by modems to indicate that the telephone line is ringing.

VI\_ATTR\_ASRL\_REPLACE\_CHAR This attribute specifies the character to be used to replace incoming characters that arrive with errors (such as parity error).

VI\_ATTR\_ASRL\_XON\_CHAR This attribute specifies the value of the XON character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.

VI\_ATTR\_ASRL\_XOFF\_CHAR This attribute specifies the value of the XOFF character used for XON/XOFF flow control (both directions). If XON/XOFF flow control (software handshaking) is not being used, the value of this attribute is ignored.

**TCPIP Specific INSTR Resource Attributes**

VI\_ATTR\_TCPIP\_ADDR This is the TCPIP address of the device to which the session is connected. This string is formatted in dot-notation.

VI\_ATTR\_TCPIP\_HOSTNAME This specifies the host name of the device. If no host name is available, this attribute returns an empty string.

VI\_ATTR\_TCPIP\_DEVICE\_NAME This specifies the LAN device name used by the VXI-11 or HiSLIP protocol during connection.

VI\_ATTR\_TCPIP\_IS\_HISLIP Specifies whether this resource uses the HiSLIP protocol.

**VXI, GPIB-VXI, and USB Specific INSTR Resource Attributes**

VI\_ATTR\_4882\_COMPLIANT Specifies whether the device is 488.2 compliant.

**VXI, GPIB-VXI, USB, and PXI Specific INSTR Resource Attributes**

VI\_ATTR\_ManF\_Id Manufacturer identification number of the device. For PXI, if Subsystem ID and Subsystem Vendor ID are defined for the device, then this attribute value is the Subsystem Vendor ID, or else this attribute value is the PCI Vendor ID.

VI\_ATTR\_Model\_Code Model code for the device. For PXI, If Subsystem ID and Subsystem Vendor ID are defined for the device, then this attribute value is the Subsystem ID, or else this attribute value is the PCI Device ID.

VI\_ATTR\_MANF\_NAME This string attribute is the manufacturer’s name. The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can be different between VISA implementations and/or revisions.

VI\_ATTR\_MODEL\_NAME This string attribute is the model name of the device. The value of this attribute should be used for display purposes only and not for programmatic decisions, as the value can be different between VISA implementations and/or revisions.

**USB Specific INSTR Resource Attributes**

VI\_ATTR\_USB\_SERIAL\_NUM This string attribute is the serial number of the USB instrument. The value of this attribute should be used for display purposes only and not for programmatic decisions.

VI\_ATTR\_USB\_INTFC\_NUM Specifies the USB interface number of this device to which this session is connected.

VI\_ATTR\_USB\_MAX\_INTR\_SIZE Specifies the maximum number of bytes that this USB device will send on the interrupt IN pipe. The default value is the same as the maximum packet size of the interrupt IN pipe.

VI\_ATTR\_USB\_PROTOCOL Specifies the USB protocol number.

**VXI, GPIB-VXI, and PXI Specific INSTR Resource Attributes**

VI\_ATTR\_SLOT Physical slot location of the device. If the slot number is not known, VI\_UNKNOWN\_SLOT is returned.

VI\_ATTR\_SRC\_INCREMENT This is used in the viMoveIn*XX*() operation to specify how much the source offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveIn*XX*() operation moves from consecutive elements. If this attribute is set to 0, the viMoveIn*XX*() operation will always read from the same element, essentially treating the source as a FIFO register.

VI\_ATTR\_DEST\_INCREMENT This is used in the viMoveOut*XX*() operation to specify how much the destination offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOut*XX*() operation moves into consecutive elements. If this attribute is set to 0, the viMoveOut*XX*() operation will always write to the same element, essentially treating the destination as a FIFO register.

VI\_ATTR\_WIN\_ACCESS Modes in which the current window may be accessed: not currently mapped, through operations viPeek*XX*() and viPoke*XX*() only, or through operations and/or by directly dereferencing the address parameter as a pointer.

VI\_ATTR\_WIN\_BASE\_ADDR\_64

VI\_ATTR\_WIN\_BASE\_ADDR\_32 Base address of the interface bus to which this window is mapped.

VI\_ATTR\_WIN\_SIZE\_64

VI\_ATTR\_WIN\_SIZE\_32 Size of the region mapped to this window.

**PXI Specific INSTR Resource Attributes**

VI\_ATTR\_PXI\_BUS\_NUM PCI bus number of this device.

VI\_ATTR\_PXI\_DEV\_NUM PCI device number of this device.

VI\_ATTR\_PXI\_FUNC\_NUM PCI function number of the device. All devices have a function 0. Multifunction devices will also support other function numbers.

VI\_ATTR\_PXI\_SLOTPATH Slot path of this device. PXI slot paths are a sequence of values representing the PCI device number and function number of a PCI module and each parent PCI bridge that routes the module to the host PCI bridge. The string format of the attribute value is device1[.function1][,device2[.function2]][,...].

VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT Slot number or special feature connected to the local bus left lines of this device.

VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT Slot number or special feature connected to the local bus right lines of this device.

VI\_ATTR\_PXI\_TRIG\_BUS Number of the trigger bus connected to this device in the chassis.

VI\_ATTR\_PXI\_STAR\_TRIG\_BUS Number of the star trigger bus connected to this device in the chassis.

VI\_ATTR\_PXI\_STAR\_TRIG\_LINE PXI\_STAR line connected to this device.

VI\_ATTR\_PXI\_MEM\_TYPE\_BARn Memory type (memory mapped or I/O mapped) used by the device in the specified BAR.

VI\_ATTR\_PXI\_MEM\_BASE\_BARn\_32

VI\_ATTR\_PXI\_MEM\_BASE\_BARn\_64 Memory base address assigned to the specified BAR for this device.

VI\_ATTR\_PXI\_MEM\_SIZE\_BARn\_32

VI\_ATTR\_PXI\_MEM\_SIZE\_BARn\_64Size of the memory assigned to the specified BAR for this device.

VI\_ATTR\_PXI\_CHASSIS Chassis number in which this device is located.

VI\_ATTR\_PXI\_IS\_EXPRESS Specifies whether this device is PXI Express.

VI\_ATTR\_PXI\_SLOT\_LWIDTH Specifies the link width used by the slot in which this device is located.

VI\_ATTR\_PXI\_MAX\_LWIDTH Specifies the maximum link width that this device can use.

VI\_ATTR\_PXI\_ACTUAL\_LWIDTH Specifies the negotiated link width that this device is using.

VI\_ATTR\_PXI\_DSTAR\_BUS Number of the DSTAR bus connected to this device in the chassis.

VI\_ATTR\_PXI\_DSTAR\_SET Specifies the set of PXI\_DSTAR lines connected to this device.

VI\_ATTR\_PXI\_ALLOW\_WRITE\_COMBINE Specifies whether the implementation should attempt to combine bus write transfers into a larger transfer before bursting over the PCI bus.

**HiSLIP Specific INSTR Resource Attributes**

VI\_ATTR\_TCPIP\_HISLIP\_OVERLAP\_EN This enables HiSLIP ‘Overlap’ mode and its value defaults to the mode suggested by the instrument on HiSLIP connection. If disabled, the connection uses ‘Synchronous’ mode to detect and recover from interrupted errors. If enabled, the connection uses ‘Overlapped’ mode to allow overlapped responses. If changed, VISA will do a Device Clear operation to change the mode.

VI\_ATTR\_TCPIP\_HISLIP\_VERSION This is the HiSLIP protocol version used for a particular HiSLIP connetion. Currently, HiSLIP version 1.0 would return a ViVersion value of 0x00100000.

VI\_ATTR\_TCPIP\_HISLIP\_MAX\_MESSAGE\_KB This is the maximum HiSLIP message size VISA will accept from a HiSLIP system in units of kilobytes (1024 bytes). Defaults to 1024 (a 1 MB maximum message size).

**RULE 5.1.11**

All INSTR resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_TMO\_VALUE, VI\_ATTR\_INTF\_NUM, VI\_ATTR\_TRIG\_ID, and VI\_ATTR\_DMA\_ALLOW\_EN.

**RULE 5.1.12**

An INSTR resource implementation for a GPIB, GPIB-VXI, VXI, ASRL, TCPIP, or USB system **SHALL** support the attributes VI\_ATTR\_IO\_PROT, VI\_ATTR\_SEND\_END\_EN, VI\_ATTR\_SUPPRESS\_END\_EN, VI\_ATTR\_TERMCHAR, VI\_ATTR\_TERM\_CHAR\_EN, VI\_ATTR\_RD\_BUF\_OPER\_MODE, VI\_ATTR\_WR\_BUF\_OPER\_MODE, and VI\_ATTR\_FILE\_APPEND\_EN.

**RULE 5.1.13**

An INSTR resource implementation for a GPIB or GPIB-VXI system **SHALL** support the attributes VI\_ATTR\_GPIB\_PRIMARY\_ADDR, VI\_ATTR\_GPIB\_SECONDARY\_ADDR, VI\_ATTR\_GPIB\_READDR\_EN, VI\_ATTR\_GPIB\_UNADDR\_EN, and VI\_ATTR\_GPIB\_REN\_STATE.

**RULE 5.1.14**

An INSTR resource implementation for a VXI or GPIB-VXI system **SHALL** support the attributes VI\_ATTR\_FDC\_CHNL, VI\_ATTR\_FDC\_MODE, VI\_ATTR\_MEM\_BASE, VI\_ATTR\_MEM\_SIZE, VI\_ATTR\_MEM\_SPACE, VI\_ATTR\_SLOT, VI\_ATTR\_VXI\_LA, VI\_ATTR\_CMDR\_LA, VI\_ATTR\_WIN\_BASE\_ADDR, VI\_ATTR\_WIN\_SIZE, VI\_ATTR\_MAINFRAME\_LA, VI\_ATTR\_FDC\_USE\_PAIR, VI\_ATTR\_FDC\_GEN\_SIGNAL\_EN, VI\_ATTR\_SRC\_INCREMENT, VI\_ATTR\_DEST\_INCREMENT, VI\_ATTR\_WIN\_ACCESS, VI\_ATTR\_IMMEDIATE\_SERV, VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, VI\_ATTR\_WIN\_BYTE\_ORDER, VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, VI\_ATTR\_WIN\_ACCESS\_PRIV, VI\_ATTR\_VXI\_DEV\_CLASS, and VI\_ATTR\_VXI\_TRIG\_SUPPORT.

**RULE 5.1.15**

An INSTR resource implementation for an ASRL system **SHALL** support the attributes VI\_ATTR\_ASRL\_BAUD, VI\_ATTR\_ASRL\_DATA\_BITS, VI\_ATTR\_ASRL\_PARITY, VI\_ATTR\_ASRL\_STOP\_BITS, VI\_ATTR\_ASRL\_FLOW\_CNTRL, VI\_ATTR\_ASRL\_END\_IN, VI\_ATTR\_ASRL\_END\_OUT, VI\_ATTR\_ASRL\_REPLACE\_CHAR, VI\_ATTR\_ASRL\_XON\_CHAR, and VI\_ATTR\_ASRL\_XOFF\_CHAR.

**RULE 5.1.16**

An INSTR resource implementation for a TCPIP system **SHALL** support the attributes VI\_ATTR\_TCPIP\_ADDR, VI\_ATTR\_TCPIP\_HOSTNAME, VI\_ATTR\_TCPIP\_IS\_HISLIP, and VI\_ATTR\_TCPIP\_DEVICE\_NAME.

**RULE 5.1.17**

An INSTR resource implementation for a HiSLIP TCPIP system **SHALL** support the attributes VI\_ATTR\_TCPIP\_PORT, VI\_ATTR\_TCPIP\_NODELAY, VI\_ATTR\_TCPIP\_KEEPALIVE, VI\_ATTR\_TCPIP\_HISLIP\_OVERLAP\_EN, VI\_ATTR\_TCPIP\_HISLIP\_VERSION, and VI\_ATTR\_TCPIP\_HISLIP\_MAX\_MESSAGE\_KB.

**RULE 5.1.18**

For each INSTR session, the attribute VI\_ATTR\_TRIG\_ID **SHALL** be R/W (readable and writeable) when the corresponding session is not enabled for sensing triggers (via viEnableEvent() for trigger events).

**RULE 5.1.19**

For each INSTR session, the attribute VI\_ATTR\_TRIG\_ID **SHALL** be RO (read only and not writeable) when the corresponding session is enabled for sensing triggers (via viEnableEvent() for trigger events).

**RULE 5.1.20**

**IF** a GPIB or GPIB-VXI INSTR resource does not have an associated GPIB secondary address, **THEN** the call to viGetAttribute() **SHALL** return the completion code VI\_SUCCESS and the value of the attribute returned **SHALL** be VI\_NO\_SEC\_ADDR.

**RULE 5.1.21**

**IF** a GPIB or GPIB-VXI INSTR resource does not support HS488 data transfers, **AND** the attribute is VI\_ATTR\_IO\_PROT, **AND** the attribute state is VI\_PROT\_HS488, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

**OBSERVATION 5.1.2**

RULE 5.2.8 allows the HS488 protocol as an optional attribute range value for GPIB and   
GPIB-VXI INSTR resources.

**PERMISSION** **5.1.1**

**IF** the attribute VI\_ATTR\_IMMEDIATE\_SERV for a given VXI or GPIB-VXI INSTR is VI\_FALSE, **THEN** calls to viRead(), viReadAsync(), viWrite(), viWriteAsync(), viAssertTrigger(), viReadSTB(), and viClear() on sessions to the given INSTR resource **MAY** return VI\_ERROR\_NSUP\_OPER.

**PERMISSION** **5.1.2**

**IF** the range value of 0 is passed to viSetAttribute() for VI\_ATTR\_SRC\_INCREMENT or VI\_ATTR\_DEST\_INCREMENT, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**RULE 5.1.22**

**IF** a GPIB or GPIB-VXI INSTR resource does not support turning off device readdressing, **AND** the attribute is VI\_ATTR\_GPIB\_READDR\_EN, **AND** the attribute state is VI\_FALSE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

**OBSERVATION 5.1.3**

RULE 5.1.20 allows disabling unnecessary device readdressing using an optional attribute range value for GPIB and GPIB-VXI resources.

**RULE 5.1.23**

An INSTR resource implementation for a VXI or GPIB-VXI system **SHALL** support the attribute state VI\_BIG\_ENDIAN for the attributes VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, and VI\_ATTR\_WIN\_BYTE\_ORDER.

**PERMISSION** **5.1.3**

**IF** the range value of VI\_LITTLE\_ENDIAN is passed to viSetAttribute() for VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, or VI\_ATTR\_WIN\_BYTE\_ORDER, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**OBSERVATION 5.1.4**

As an example of VI\_BIG\_ENDIAN and VI\_LITTLE\_ENDIAN formats, assume that the data 0x12 is at VXI address 0, 0x34 is at address 1, 0x56 at 2, and 0x78 at 3. A 32-bit access at address 0 using VI\_BIG\_ENDIAN format would return 0x12345678; the same access using VI\_LITTLE\_ENDIAN format would return 0x78563412. Notice that the setting of the attribute values has no relation to and no effect on the native byte order of the local machine.

**RULE 5.1.24**

An INSTR resource implementation for a VXI or GPIB-VXI system **SHALL** support the attribute state VI\_DATA\_PRIV for the attributes VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, and VI\_ATTR\_WIN\_ACCESS\_PRIV.

**PERMISSION** **5.1.4**

**IF** any range value other than VI\_DATA\_PRIV is passed to viSetAttribute() for VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, or VI\_ATTR\_WIN\_ACCESS\_PRIV, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**OBSERVATION 5.1.5**

Other access privilege enumeration values may require hardware support that is not implemented. For example, the VI\_D64\_SST\* values are only supported on VXI-1 4.0-compliant controllers.

**RULE 5.1.25**

**IF** a VISA system implements the INSTR resource for a VXI system, **THEN** it **SHALL** implement the MEMACC resource for a VXI system.

**RULE 5.1.26**

**IF** a VISA system implements the INSTR resource for a GPIB-VXI system, **THEN** it **SHALL** implement the MEMACC resource for a GPIB-VXI system.

**RULE 5.1.27**

For VISA 2.2, the attributes VI\_ATTR\_WIN\_ACCESS\_PRIV and VI\_ATTR\_WIN\_BYTE\_ORDER are R/W (readable and writeable) when the corresponding session is not mapped (VI\_ATTR\_WIN\_ACCESS == VI\_NMAPPED).

**RULE 5.1.28**

For VISA 2.2, the attributes VI\_ATTR\_WIN\_ACCESS\_PRIV and VI\_ATTR\_WIN\_BYTE\_ORDER are RO (read-only) when the corresponding session is mapped (VI\_ATTR\_WIN\_ACCESS != VI\_NMAPPED).

**RULE 5.1.29**

An INSTR resource implementation for a TCPIP system **SHALL** use VXI-11 protocol for INSTR resource descriptors containing device names starting with ‘vxi’, ‘gpib’, or ‘inst’.

**RULE 5.1.30**

An INSTR resource implementation for a TCPIP system **SHALL** use HiSLIP protocol for INSTR resource descriptors containing device names starting with ‘hislip’.

**RULE 5.1.31**

**IF** an INSTR resource descriptor contains no device name forcing the protocol choice, **THEN** viOpen() **SHALL** attempt a VXI-11 connection first, **AND IF** the VXI-11 attempt connection fails, **THEN** viOpen() **SHALL** attempt a HiSLIP connection.

**PERMISSION** **5.1.5**

**IF** an INSTR resource descriptor contains no device name forcing the protocol choice, **THEN** a VISA implementation **MAY** permit configuration outside the VISA API to try a HiSLIP connection first.

**RULE 5.1.32**

**IF** an INSTR resource implementation does not support DMA transfers, **AND** the attribute is VI\_ATTR\_DMA\_ALLOW\_EN, **AND** the attribute state is VI\_TRUE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

**RULE 5.1.33**

An INSTR resource implementation for a PXI system **SHALL** use the plug-in mechanism defined in the IVI‑6.3 specification for detecting and accessing PXI devices.

**RULE 5.1.34**

**IF** a PXI INSTR resource does not support write combining, **AND** the attribute is VI\_ATTR\_PXI\_ALLOW\_WRITE\_COMBINE, **AND** the attribute state is VI\_TRUE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

**OBSERVATION 5.1.6**

It is valid for a PXI INSTR session to have both VI\_ATTR\_PXI\_ALLOW\_WRITE\_COMBINE and VI\_ATTR\_DMA\_ALLOW\_EN set to VI\_TRUE. In this case, write combining is enabled for the viMoveOut() functions, whereas DMA is enabled for the viMoveIn() functions.

**RULE 5.1.35**

An INSTR resource implementation for a USB system **SHALL** use the protocol defined in the USB Test and Measurement class (USBTMC) specification or a USBTMC subclass specification.

**RULE 5.1.36**

An INSTR resource implementation for a USB system **SHALL** support the value of VI\_TRUE for the attribute VI\_ATTR\_TERMCHAR\_EN even if the USB interface does not indicate support for TermChar in its capabilities bits.

**OBSERVATION 5.1.7**

A given VISA implementation of an INSTR resource for a USB system can choose how to implement termination character support if the device does not support it natively. Two possible valid options are for the VISA implementation to request 1 byte at a time from the device, or for the VISA implementation to request larger blocks of data and buffer the data internally.

**RULE 5.1.37**

An INSTR resource implementation for a VXI or GPIB-VXI or USB system **SHALL** support the attributes VI\_ATTR\_MANF\_ID, VI\_ATTR\_MODEL\_CODE, VI\_ATTR\_MANF\_NAME, VI\_ATTR\_MODEL\_NAME, and VI\_ATTR\_4882\_COMPLIANT.

**RULE 5.1.38**

An INSTR resource implementation for a USB system **SHALL** support the attributes VI\_ATTR\_USB\_SERIAL\_NUM, VI\_ATTR\_USB\_INTFC\_NUM, VI\_ATTR\_USB\_MAX\_INTR\_SIZE, and VI\_ATTR\_USB\_PROTOCOL.

**RULE 5.1.39**

For each INSTR session, the attribute VI\_ATTR\_USB\_MAX\_INTR\_SIZE **SHALL** be R/W (readable and writeable) when the corresponding session is not enabled for sensing USB interrupts (via viEnableEvent() for USB interrupt events).

**RULE 5.1.40**

For each INSTR session, the attribute VI\_ATTR\_USB\_MAX\_INTR\_SIZE **SHALL** be RO (read only and not writeable) when the corresponding session is enabled for sensing USB interrupts (via viEnableEvent() for USB interrupt events).

**OBSERVATION 5.1.8**

In a previous version of the VISA specification, the I/O protocol value names were VI\_NORMAL, VI\_FDC, VI\_HS488, and VI\_ASRL488. The new names are VI\_PROT\_NORMAL, VI\_PROT\_FDC, VI\_PROT\_HS488, and VI\_PROT\_4882\_STRS. It is the intent of this specification that the numeric values for these names must be consistent for backward compatibility.

**RULE 5.1.41**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_MEM\_SIZE and VI\_ATTR\_MEM\_SIZE\_64 **SHALL** be identical.

**RULE 5.1.42**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_MEM\_BASE and VI\_ATTR\_MEM\_BASE\_32 **SHALL** be identical.

**RULE 5.1.43**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_MEM\_BASE and VI\_ATTR\_MEM\_BASE\_64 **SHALL** be identical.

**RULE 5.1.44**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_MEM\_SIZE and VI\_ATTR\_MEM\_SIZE\_32 **SHALL** be identical.

**RULE 5.1.45**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_WIN\_BASE\_ADDR and VI\_ATTR\_WIN\_BASE\_ADDR\_32 **SHALL** be identical.

**RULE 5.1.46**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_WIN\_BASE\_ADDR and VI\_ATTR\_WIN\_BASE\_ADDR\_64 **SHALL** be identical.

**RULE 5.1.47**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_WIN\_SIZE and VI\_ATTR\_WIN\_SIZE\_32 **SHALL** be identical.

**RULE 5.1.48**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_WIN\_SIZE and VI\_ATTR\_WIN\_SIZE\_64 **SHALL** be identical.

**RULE 5.1.49**

**IF** a user calls viGetAttribute() with the attribute VI\_ATTR\_MEM\_BASE\_32 and the value would not fit in a 32-bit integer (meaning the value is greater than 0xFFFFFFFF), **THEN** the implementation **SHALL** return VI\_ERROR\_NSUP\_OFFSET.

**OBSERVATION 5.1.9**

When the VXI memory base fits in a 32-bit integer, calling viGetAttribute() with the attributes VI\_ATTR\_MEM\_BASE\_32 and VI\_ATTR\_MEM\_BASE\_64 return the same status and value.

**RULE 5.1.50**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n* and VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_32 **SHALL** be identical.

**RULE 5.1.51**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n* and VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_64 **SHALL** be identical.

**RULE 5.1.52**

**IF** a user calls viGetAttribute() with the attribute VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_32 and the value would not fit in a 32-bit integer (meaning the value is greater than 0xFFFFFFFF), **THEN** the implementation **SHALL** return VI\_ERROR\_NSUP\_OFFSET.

**OBSERVATION 5.1.10**

When the PXI memory base fits in a 32-bit integer, calling viGetAttribute() with the attributes VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_32 and VI\_ATTR\_PXI\_MEM\_BASE\_BAR*n*\_64 return the same status and value.

5.1.3 INSTR Resource Events

This resource defines the following events for communication with applications.

**VI\_EVENT\_SERVICE\_REQ**

**Description**

Notification that a service request was received from the device.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_SERVICE\_REQ |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_VXI\_SIGP**

**Description**

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

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_VXI\_SIGP |
| VI\_ATTR\_SIGP\_STATUS\_ID | RO | ViUInt16 | 0 to FFFFh |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_SIGP\_STATUS\_ID The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.

**VI\_EVENT\_TRIG**

**Description**

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

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_TRIG |
| VI\_ATTR\_RECV\_TRIG\_ID | RO | ViInt16 | VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_INSTR |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_RECV\_TRIG\_ID The identifier of the triggering mechanism on which the specified trigger event was received.

**VI\_EVENT\_IO\_COMPLETION**

**Description**

Notification that an asynchronous operation has completed.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_IO\_COMPLETION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_JOB\_ID | RO | ViJobId | N/A |
| VI\_ATTR\_BUFFER | RO | ViBuf | N/A |
| VI\_ATTR\_RET\_COUNT | RO | ViBusSize | \* |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |
| VI\_ATTR\_RET\_COUNT\_32 | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RET\_COUNT\_64\*\* | RO | ViUInt64 | 0 to FFFFFFFFFFFFFFFFh |

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

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

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS This field contains the return code of the asynchronous I/O operation that has completed.

VI\_ATTR\_JOB\_ID This field contains the job ID of the asynchronous operation that has completed.

VI\_ATTR\_BUFFER This field contains the address of a buffer that was used in an asynchronous operation.

VI\_ATTR\_RET\_COUNT This field contains the actual number of elements that were

VI\_ATTR\_RET\_COUNT\_32 asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_64

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

For more information on VI\_ATTR\_OPER\_NAME, see its definition in Section 3.7.2.3, *VI\_EVENT\_EXCEPTION*.

**VI\_EVENT\_VXI\_VME\_INTR**

**Description**

Notification that a VXIbus interrupt was received from the device.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_VXI\_VME\_INTR |
| VI\_ATTR\_INTR\_STATUS\_ID | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RECV\_INTR\_LEVEL | RO | ViInt16 | 1 to 7, VI\_UNKNOWN\_LEVEL |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_INTR\_STATUS\_ID This attribute value is the 32-bit status/ID retrieved during the IACK cycle.

VI\_ATTR\_RECV\_INTR\_LEVEL This attribute value is the VXI interrupt level on which the interrupt was received.

**VI\_EVENT\_USB\_INTR**

**Description**

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

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_USB\_INTR |
| VI\_ATTR\_USB\_RECV\_INTR\_SIZE | RO | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_USB\_RECV\_INTR\_DATA | RO | ViBuf | N/A |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_USB\_RECV\_INTR\_SIZE Specifies the size of the data that was received from the USB interrupt-IN pipe. This value will never be larger than the session’s value of VI\_ATTR\_USB\_MAX\_INTR\_SIZE.

VI\_ATTR\_USB\_RECV\_INTR\_DATA Specifies the actual data that was received from the USB interrupt-IN pipe. Querying this attribute copies the contents of the data to the user’s buffer. The user’s buffer must be sufficiently large enough to hold all of the data.

VI\_ATTR\_STATUS Specifies the status of the read operation from the USB interrupt-IN pipe. If the device sent more data than the user specified in VI\_ATTR\_USB\_MAX\_INTR\_SIZE, then this attribute value will contain the status code VI\_WARN\_QUEUE\_OVERFLOW.

VI\_EVENT\_PXI\_INTR

**Description**

Notification that a PCI Interrupt was received from the device.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| Symbolic Name | Access Privilege | Data Type | Range |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_PXI\_INTR |
| VI\_ATTR\_PXI\_RECV\_INTR\_SEQ | RO | ViInt16 | N/A |
| VI\_ATTR\_PXI\_RECV\_INTR\_DATA | RO | ViUInt32 | N/A |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_PXI\_RECV\_INTR\_SEQ Specifies the index of the interrupt sequence that detected the interrupt condition.

VI\_ATTR\_PXI\_RECV\_INTR\_DATA Specifies the first PXI/PCI register that was read in the successful interrupt detection sequence.

**RULE 5.1.53**

All INSTR resource implementations **SHALL** support the generation of the events VI\_EVENT\_IO\_COMPLETION and VI\_EVENT\_EXCEPTION.

**RULE 5.1.54**

An INSTR resource implementation for a GPIB, GPIB-VXI, VXI, TCPIP, or USB system **SHALL** support the generation of the event VI\_EVENT\_SERVICE\_REQ.

**RULE 5.1.55**

An INSTR resource implementation for a VXI system **SHALL** support the generation of the events VI\_EVENT\_VXI\_SIGP, VI\_EVENT\_TRIG, and VI\_EVENT\_VXI\_VME\_INTR.

**RULE 5.1.56**

An INSTR resource implementation for a PXI system **SHALL** support the generation of the event VI\_EVENT\_PXI\_INTR.

**RULE 5.1.57**

On some operating systems, it may be a requirement to handle PXI interrupts in the OS kernel environment. VISA implementations on such operating systems **SHALL** provide a mechanism for performing device-specific operations in the kernel in response to an interrupt. The PXI Module Description File Specification specifies a VISA Registration Descriptor for this purpose. This mechanism allows the event to be delivered to the instrument driver software in the application environment once the PXI interrupt has been safely removed in the OS kernel environment.

To implement the above rule, a VISA implementation could implement the following behavior.

1. The user, integrator, or instrument driver developer registers information from the module description file with the VISA implementation. The information about the device registered includes a description of these operations:
2. How to detect whether the device is asserting a PXI interrupt (Operation DETECT).
3. How to stop the device from asserting its PXI interrupt line. (Operation QUIESCE).
4. When the user enables events from the device, the VISA implementation reads the device description to find descriptions of the above operations.
5. Upon receiving an interrupt, the VISA implementation uses OS services combined with the DETECT operation on each device to determine which device is interrupting.
6. The VISA implementation uses the QUIESCE operation on the interrupting device.
7. The VISA implementation delivers the VI\_EVENT\_PXI\_INTR to each session enabled for interrupts to that device.

**OBSERVATION 5.1.11**

In any implementation, the VISA client code must ensure that the device is enabled to drive the interrupt line again after handling the condition that caused the interrupt.

**RULE 5.1.58**

**IF** a session is enabled for VI\_EVENT\_VXI\_SIGP, **AND** a VXI interrupt or signal is detected with the value FD*xx* (where *xx* is the logical address associated with the given session), **THEN** the VISA system **SHALL** generate a VI\_EVENT\_VXI\_SIGP in addition to a VI\_EVENT\_SERVICE\_REQ.

**RULE 5.1.59**

**IF** a session is enabled for VI\_EVENT\_VXI\_VME\_INTR, **AND** a VXI interrupt is detected with the value FD*xx* (where *xx* is the logical address associated with the given session), **THEN** the VISA system **SHALL** generate a VI\_EVENT\_VXI\_VME\_INTR in addition to a VI\_EVENT\_SERVICE\_REQ.

**RULE 5.1.60**

An INSTR resource implementation for a VXI or GPIB-VXI system **SHALL** return the error VI\_ERROR\_INV\_EVENT when a user tries to enable VI\_EVENT\_SERVICE\_REQ for VME devices or VXI register based devices.

**RULE 5.1.61**

An INSTR resource implementation for a USB system **SHALL** return the error VI\_ERROR\_INV\_EVENT when a user tries to enable VI\_EVENT\_SERVICE\_REQ for USBTMC base-class (non-488) devices.

**RULE 5.1.62**

An INSTR resource implementation for a USB system **SHALL** return the error VI\_ERROR\_INV\_EVENT when a user tries to enable VI\_EVENT\_SERVICE\_REQ for a USB488 device that does not have an interrupt IN pipe.

**RULE 5.1.63**

An INSTR resource implementation for a USB system **SHALL** support the generation of the event VI\_EVENT\_USB\_INTR.

**RULE 5.1.64**

An INSTR resource implementation for a USB system **SHALL** return the error VI\_ERROR\_INV\_EVENT when a user tries to enable VI\_EVENT\_USB\_INTR for a USBTMC device (base-class or USB488) that does not have an interrupt IN pipe.

**RULE 5.1.65**

An INSTR resource implementation for a USB system **SHALL** generate VI\_EVENT\_USB\_INTR only when the interrupt header contains a vendor-specific notification as defined by the USBTMC specification.

**OBSERVATION 5.1.12**

A USB488 service request notification will not cause VI\_EVENT\_USB\_INTR to be generated.

**RULE 5.1.66**

**IF** a framework is 32-bit, **THEN** the values of the attributes VI\_ATTR\_RET\_COUNT and VI\_ATTR\_RET\_COUNT\_32 **SHALL** be identical.

**RULE 5.1.67**

**IF** a framework is 64-bit, **THEN** the values of the attributes VI\_ATTR\_RET\_COUNT and VI\_ATTR\_RET\_COUNT\_64 **SHALL** be identical.

**RULE 5.1.68**

**IF** a framework is 32-bit, **THEN** the attribute VI\_ATTR\_RET\_COUNT\_64 **SHALL NOT** be defined.

**OBSERVATION 5.1.13**

A user on a 32-bit framework cannot transfer more data than would fit in a 32-bit size.

5.1.4 INSTR Resource Operations

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, fileName, count, retCount)

viAssertTrigger(vi, protocol)

viReadSTB(vi, status)

viClear(vi)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viBufRead(vi, buf, count, retCount)

viQueryf(vi, writeFmt, readFmt, arg1, arg2, ...)

viVQueryf(vi, writeFmt, readFmt, params)

viIn8(vi, space, offset, val8)

viIn16(vi, space, offset, val16)

viIn32(vi, space, offset, val32)

viIn64(vi, space, offset, val64)

viOut8(vi, space, offset, val8)

viOut16(vi, space, offset, val16)

viOut32(vi, space, offset, val32)

viOut64(vi, space, offset, val64)

viMoveIn8(vi, space, offset, length, buf8)

viMoveIn16(vi, space, offset, length, buf16)

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

viMoveIn64(vi, space, offset, length, buf64)

viMoveOut8(vi, space, offset, length, buf8)

viMoveOut16(vi, space, offset, length, buf16)

viMoveOut32(vi, space, offset, length, buf32)

viMoveOut64(vi, space, offset, length, buf64)

viMoveIn8Ex(vi, space, offset64, length, buf8)

viMoveIn16Ex(vi, space, offset64, length, buf16)

viMoveIn32Ex(vi, space, offset64, length, buf32)

viMoveIn64Ex(vi, space, offset64, length, buf64)

viMoveOut8Ex(vi, space, offset64, length, buf8)

viMoveOut16Ex(vi, space, offset64, length, buf16)

viMoveOut32Ex(vi, space, offset64, length, buf32)

viMoveOut64Ex(vi, space, offset64, length, buf64)

viMove(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

viMoveAsync(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length)

viMoveAsyncEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId)

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

viMapAddressEx(vi, mapSpace, mapBase64, mapSize, access, suggested, address)

viUnmapAddress(vi)

viPeek8(vi, addr, val8)

viPeek16(vi, addr, val16)

viPeek32(vi, addr, val32)

viPeek64(vi, addr, val64)

viPoke8(vi, addr, val8)

viPoke16(vi, addr, val16)

viPoke32(vi, addr, val32)

viPoke64(vi, addr, val64)

viMemAlloc(vi, size, offset)

viMemFree(vi, offset)

viMemAllocEx(vi, size, offset64)

viMemFreeEx(vi, offset64)

viGpibControlREN(vi, mode)

viVxiCommandQuery(vi, mode, cmd, response)

viUsbControlOut(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf)

viUsbControlIn(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf, retCnt)

**RULE 5.1.69**

An INSTR resource implementation for a GPIB system **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(), viWrite(), viWriteAsync(), viWriteFromFile(), viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viScanf(), viVScanf(), viQueryf(), viVQueryf(), viSPrintf(), viVSPrintf(), viBufWrite(), viSScanf(), viVSScanf(), viBufRead(), and viGpibControlREN().

**RULE 5.1.70**

An INSTR resource implementation for a GPIB-VXI or VXI system **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(), viWrite(), viWriteAsync(), viWriteFromFile(),viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viScanf(), viVScanf(), viQueryf(), viVQueryf(), viIn8(), viIn16(), viIn32(), viIn64(), viOut8(), viOut16(), viOut32(), viOut64(), viMoveIn8(), viMoveIn16(), viMoveIn32(), viMoveIn64(), viMoveOut8(), viMoveOut16(), viMoveOut32(), viMoveOut64(), viMoveIn8Ex(), viMoveIn16Ex(), viMoveIn32Ex(), viMoveIn64Ex(), viMoveOut8Ex(), viMoveOut16Ex(), viMoveOut32Ex(), viMoveOut64Ex(), viMoveAsync(), viMapAddress(), viMoveAsyncEx(), viMapAddressEx(), viUnmapAddress(), viPeek8(), viPeek16(), viPeek32(), viPeek64(), viPoke8(), viPoke16(), viPoke32(), viPoke64(), viMemAlloc(), viMemFree(), viMemAllocEx(), viMemFreeEx(), viSPrintf(), viVSPrintf(), viBufWrite(), viSScanf(), viVSScanf(), viBufRead(), and viVxiCommandQuery().

**RULE 5.1.71**

An INSTR resource implementation for an ASRL system **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(),viWrite(), viWriteAsync(), viWriteFromFile(), viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viScanf(), viVScanf(), viQueryf(), viVQueryf(), viSPrintf(), viVSPrintf(), viBufWrite(), viSScanf(), viVSScanf(), and viBufRead().

**RULE 5.1.72**

An INSTR resource implementation for a TCPIP system **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(), viWrite(), viWriteAsync(), viWriteFromFile(), viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viScanf(), viVScanf(), viQueryf(), viVQueryf(), viSPrintf(), viVSPrintf(), viBufWrite(), viSScanf(), viVSScanf(), and viBufRead().

**RULE 5.1.73**

An INSTR resource implementation for a HiSLIP TCPIP system **SHALL** support the operation viGpibControlREN().

**RULE 5.1.74**

An INSTR resource implementation for a USB system **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(), viWrite(), viWriteAsync(), viWriteFromFile(), viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viScanf(), viVScanf(), viQueryf(), viVQueryf(), viSPrintf(), viVSPrintf(), viBufWrite(), viSScanf(), viVSScanf(), viBufRead(), viGpibControlREN(), viUsbControlOut(), and viUsbControlIn().

**RULE 5.1.75**

An INSTR resource implementation for a PXI system SHALL support the operations viAssertTrigger(), viIn8(), viIn16(), viIn32(), viIn64(), viOut8(), viOut16(), viOut32(), viOut64(), viMoveIn8(), viMoveIn16(), viMoveIn32(), viMoveIn64(), viMoveOut8(), viMoveOut16(), viMoveOut32(), viMoveOut64(), viMoveIn8Ex(), viMoveIn16Ex(), viMoveIn32Ex(), viMoveIn64Ex(), viMoveOut8Ex(), viMoveOut16Ex(), viMoveOut32Ex(), viMoveOut64Ex(), viMove(), viMoveAsync(), viMoveEx(), viMoveAsyncEx(), viMapAddress(), viMapAddressEx(), viUnmapAddress(), viPeek8(), viPeek16(), viPeek32(), viPeek64(), viPoke8(), viPoke16(), viPoke32(), and viPoke64().

5.1.5 Differences between VXI-11 and HiSLIP TCPIP INSTR Systems

While a HiSLIP system provides many VXI-11-like capabilities, it differs in several respects.

In particular, operations are sent to the HiSLIP device in ‘fire and forget’ form with immediate return, unlike VXI-11, where each operation is blocks until a VXI-11 device handshake return. HiSLIP does utilize TCP/IP to send operations, which does guarantee that HiSLIP messages are delivered in order and not lost, but this does not guarantee that the HiSLIP device has finished, or even started, the requested operation after a viWrite() call, for example.

HiSLIP systems also provide services for exclusive and shared locks held in the HiSLIP device while VXI-11 only supports exclusive locks held in the VXI-11 device.

HiSLIP detects and corrects of Interrupted errors, but can also be operated in an overlapped mode where interrupted errors are ignored but responses are sent as quickly as possible from the HiSLIP system.

Like VXI-11, HiSLIP systems support sub-instrument addressing.

5.2 Memory Access Resource

The Memory Access (MEMACC) Resource encapsulates the address space of a memory mapped bus such as the VXIbus. A VISA Memory Access Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task, such as reading a register or writing to a memory location.

5.2.1 MEMACC Resource Overview

The MEMACC Resource lets a controller interact with the interface associated with this resource. It does this by providing the controller with services to access arbitrary registers or memory addresses on memory-mapped buses. These services are described in detail in the remainder of this section.

• **Memory I/O Services**

– The High-Level Access Service allows register-level access to the interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or even VME or VXI memory through a system controlled by a GPIB-to-VXI controller. A resource exists for each interface to which the controller has access. When dealing with memory accesses, there is a tradeoff between speed and complexity, and between software overhead and encapsulation. The High-Level Access Service is similar in purpose to the Low-Level Access Service. The difference between these two services is that the High-Level Access Service has greater software overhead because it encapsulates most of the code required to perform the memory access, such as window mapping and error checking. In general, high-level accesses are slower than low-level accesses, but they encapsulate the operations necessary to perform the access and are considered safer.

The High-Level Access Service lets the programmer access memory on the interface bus through simple operations such as viIn16() and viOut16(). These operations encapsulate the map/unmap and peek/poke operations found in the Low-Level Access Service. There is no need to explicitly map the memory to a window.

– The Low-Level Access Service, like the High-Level Access Service, allows register-level access to the interfaces that support direct memory access, such as the VXIbus, VMEbus, MXIbus, or VME or VXI memory through a system controlled by a GPIB-to-VXI controller. A resource exists for each interface of this type that the controller has locally. When dealing with memory accesses, there is a tradeoff between speed and complexity and between software overhead and encapsulation. The Low-Level Access Service is similar in purpose to the High-Level Access Service. The difference between these two services is that the Low‑Level Access Service increases access speed by removing software overhead, but requires more programming effort by the user. To decrease the amount of overhead involved in the memory access, the Low-Level Access Service does not return any error information in the access operations.

Before an application can use the Low-Level Access Service on the interface bus, it must map a range of addresses using the operation viMapAddress(). Although the resource handles the allocation and operation of the window, the programmer must free the window via viUnmapAddress() when finished. This makes the window available for the system to reallocate.

**RULE 5.2.1**

**IF** an application performs viClose() on a session to a MEMACC resource with memory still mapped, **THEN** viClose() **SHALL** perform an implicit unmapping of the mapped window.

**PERMISSION 5.2.1**

A VISA implementation that supports the PXI MEMACC resource **MAY** limit accesses to that resource to permit only accesses to memory allocated by viMemAlloc().

5.2.2 MEMACC Resource Attributes

**Generic MEMACC Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | ViUInt16 | VI\_INTF\_VXI  VI\_INTF\_GPIB\_VXI  VI\_INTF\_PXI |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_DMA\_ALLOW\_EN | R/W | Local | ViBoolean | VI\_TRUE VI\_FALSE |

**VXI, GPIB-VXI, and PXI Specific MEMACC Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_SRC\_INCREMENT | R/W | Local | ViInt32 | 0 to 1 |
| VI\_ATTR\_DEST\_INCREMENT | R/W | Local | ViInt32 | 0 to 1 |
| VI\_ATTR\_WIN\_ACCESS | RO | Local | ViUInt16 | VI\_NMAPPED  VI\_USE\_OPERS  VI\_DEREF\_ADDR |
| VI\_ATTR\_WIN\_BASE\_ADDR\_32 | RO | Local | ViBusAddress | N/A |
| VI\_ATTR\_WIN\_BASE\_ADDR\_64 | RO | Local | ViBusAddress64 | N/A |
| VI\_ATTR\_WIN\_SIZE\_32 | RO | Local | ViBusSize | N/A |
| VI\_ATTR\_WIN\_SIZE\_64 | RO | Local | ViBusSize64 | N/A |

**VXI and GPIB-VXI Specific MEMACC Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_VXI\_LA | RO | Global | ViInt16 | 0 to 255 |
| VI\_ATTR\_SRC\_BYTE\_ORDER | R/W | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |
| VI\_ATTR\_DEST\_BYTE\_ORDER | R/W | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |
| VI\_ATTR\_WIN\_BYTE\_ORDER | R/W\* | Local | ViUInt16 | VI\_BIG\_ENDIAN  VI\_LITTLE\_ENDIAN |

(continues)

**VXI and GPIB-VXI Specific MEMACC Resource Attributes (Continued)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_SRC\_ACCESS\_PRIV | R/W | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV  VI\_D64\_NPRIV  VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 |
| VI\_ATTR\_DEST\_ACCESS\_PRIV | R/W | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV  VI\_D64\_NPRIV  VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 |
| VI\_ATTR\_WIN\_ACCESS\_PRIV | R/W\* | Local | ViUInt16 | VI\_DATA\_NPRIV  VI\_DATA\_PRIV  VI\_PROG\_NPRIV  VI\_PROG\_PRIV  VI\_BLCK\_NPRIV  VI\_BLCK\_PRIV |

\* For VISA 2.2, the attributes VI\_ATTR\_WIN\_BYTE\_ORDER and VI\_ATTR\_WIN\_ACCESS\_PRIV are R/W (readable and writeable) when the corresponding session is not mapped (VI\_ATTR\_WIN\_ACCESS == VI\_NMAPPED). When the session is mapped, these attributes are RO (read only).

**GPIB-VXI Specific MEMACC Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_PARENT\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | RO | Global | ViUInt16 | 0 to 30 |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | RO | Global | ViUInt16 | 0 to 31, VI\_NO\_SEC\_ADDR |

**Attribute Descriptions**

**Generic MEMACC Resource Attributes**

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_DMA\_ALLOW\_EN This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

**VXI, GPIB-VXI, and PXI Specific MEMACC Resource Attributes**

VI\_ATTR\_SRC\_INCREMENT This is used in the viMoveIn*XX*() operation to specify how much the source offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the source address will be incremented by 1 after each transfer), and the viMoveIn*XX*() operation moves from consecutive elements. If this attribute is set to 0, the viMoveIn*XX*() operation will always read from the same element, essentially treating the source as a FIFO register.

VI\_ATTR\_DEST\_INCREMENT This is used in the viMoveOut*XX*() operation to specify how much the destination offset is to be incremented after every transfer. The default value of this attribute is 1 (that is, the destination address will be incremented by 1 after each transfer), and the viMoveOut*XX*() operation moves into consecutive elements. If this attribute is set to 0, the viMoveOut*XX*() operation will always write to the same element, essentially treating the destination as a FIFO register.

VI\_ATTR\_WIN\_ACCESS Modes in which the current window may be accessed. The valid modes are as follows:

1. not currently mapped;
2. through the operations viPeek*XX*() and viPoke*XX*() only;
3. through operations and/or by directly dereferencing the address parameter as a pointer.

VI\_ATTR\_WIN\_BASE\_ADDR\_64

VI\_ATTR\_WIN\_BASE\_ADDR\_32 Base address of the interface bus to which this window is mapped.

VI\_ATTR\_WIN\_SIZE\_64

VI\_ATTR\_WIN\_SIZE\_32 Size of the region mapped to this window.

**VXI and GPIB-VXI Specific MEMACC Resource Attributes**

VI\_ATTR\_VXI\_LA Logical address of the local controller.

VI\_ATTR\_SRC\_BYTE\_ORDER This attribute specifies the byte order to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.

VI\_ATTR\_DEST\_BYTE\_ORDER This attribute specifies the byte order to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.

VI\_ATTR\_WIN\_BYTE\_ORDER This attribute specifies the byte order to be used in low-level access operations, such as viMapAddress(), viPeek*XX*() and viPoke*XX*(), when accessing the mapped window.

VI\_ATTR\_SRC\_ACCESS\_PRIV This attribute specifies the address modifier to be used in high-level access operations, such as viIn*XX*() and viMoveIn*XX*(), when reading from the source.

VI\_ATTR\_DEST\_ACCESS\_PRIV This attribute specifies the address modifier to be used in high-level access operations, such as viOut*XX*() and viMoveOut*XX*(), when writing to the destination.

VI\_ATTR\_WIN\_ACCESS\_PRIV This attribute specifies the address modifier to be used in low-level access operations, such as viMapAddress(), viPeek*XX*() and viPoke*XX*(), when accessing the mapped window.

**GPIB-VXI Specific MEMACC Attributes**

VI\_ATTR\_INTF\_PARENT\_NUM Board number of the GPIB board to which the GPIB-VXI is attached.

VI\_ATTR\_GPIB\_PRIMARY\_ADDR Primary address of the GPIB‑VXI controller used by the given session.

VI\_ATTR\_GPIB\_SECONDARY\_ADDR Secondary address of the GPIB‑VXI controller used by the given session.

**PERMISSION** **5.2.2**

**IF** the range value of 0 is passed to viSetAttribute() for VI\_ATTR\_SRC\_INCREMENT or VI\_ATTR\_DEST\_INCREMENT, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**PERMISSION** **5.2.3**

**IF** the range value of VI\_LITTLE\_ENDIAN is passed to viSetAttribute() for VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, or VI\_ATTR\_WIN\_BYTE\_ORDER, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**PERMISSION** **5.2.4**

**IF** any range value other than VI\_DATA\_PRIV is passed to viSetAttribute() for VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, or VI\_ATTR\_WIN\_ACCESS\_PRIV, **THEN** viSetAttribute() **MAY** return VI\_ERROR\_NSUP\_ATTR\_STATE.

**RULE 5.2.2**

All MEMACC resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_TMO\_VALUE, VI\_ATTR\_INTF\_NUM, and VI\_ATTR\_DMA\_ALLOW\_EN.

**RULE 5.2.3**

A MEMACC resource implementation for a VXI or GPIB-VXI system **SHALL** support the attributes VI\_ATTR\_WIN\_BASE\_ADDR, VI\_ATTR\_WIN\_SIZE, VI\_ATTR\_WIN\_ACCESS, VI\_ATTR\_SRC\_INCREMENT, VI\_ATTR\_DEST\_INCREMENT, VI\_ATTR\_SRC\_BYTE\_ORDER, VI\_ATTR\_DEST\_BYTE\_ORDER, VI\_ATTR\_WIN\_BYTE\_ORDER, VI\_ATTR\_SRC\_ACCESS\_PRIV, VI\_ATTR\_DEST\_ACCESS\_PRIV, and VI\_ATTR\_WIN\_ACCESS\_PRIV.

**RULE 5.2.4**

A MEMACC resource implementation for a PXI system **SHALL** support the attributes VI\_ATTR\_WIN\_BASE\_ADDR, VI\_ATTR\_WIN\_SIZE, VI\_ATTR\_WIN\_ACCESS, VI\_ATTR\_SRC\_INCREMENT, and VI\_ATTR\_DEST\_INCREMENT.

**RULE 5.2.5**

**IF** a MEMACC resource implementation does not support DMA transfers, **AND** the attribute is VI\_ATTR\_DMA\_ALLOW\_EN, **AND** the attribute state is VI\_TRUE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

5.2.3 MEMACC Resource Events

This resource defines the following event for communication with applications.

**VI\_EVENT\_IO\_COMPLETION**

**Description**

Notification that an asynchronous operation has completed.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_IO\_COMPLETION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_JOB\_ID | RO | ViJobId | N/A |
| VI\_ATTR\_BUFFER | RO | ViBuf | N/A |
| VI\_ATTR\_RET\_COUNT | RO | ViBusSize | \* |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |
| VI\_ATTR\_RET\_COUNT\_32 | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RET\_COUNT\_64\*\* | RO | ViUInt64 | 0 to FFFFFFFFFFFFFFFFh |

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

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

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS This field contains the return code of the asynchronous I/O operation that has completed.

VI\_ATTR\_JOB\_ID This field contains the job ID of the asynchronous operation that has completed.

VI\_ATTR\_BUFFER This field contains the address of a buffer that was used in an asynchronous operation.

VI\_ATTR\_RET\_COUNT This field contains the actual number of elements that were

VI\_ATTR\_RET\_COUNT\_32 asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_64

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

For more information on VI\_ATTR\_OPER\_NAME, see its definition in Section 3.7.2.3, *VI\_EVENT\_EXCEPTION*.

**RULE 5.2.6**

All MEMACC resource implementations **SHALL** support the generation of the events VI\_EVENT\_IO\_COMPLETION and VI\_EVENT\_EXCEPTION.

5.2.4 MEMACC Resource Operations

viIn8(vi, space, offset, val8)

viIn16(vi, space, offset, val16)

viIn32(vi, space, offset, val32)

viIn64(vi, space, offset, val64)

viOut8(vi, space, offset, val8)

viOut16(vi, space, offset, val16)

viOut32(vi, space, offset, val32)

viOut64(vi, space, offset, val64)

viMoveIn8(vi, space, offset, length, buf8)

viMoveIn16(vi, space, offset, length, buf16)

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

viMoveIn64(vi, space, offset, length, buf64)

viMoveOut8(vi, space, offset, length, buf8)

viMoveOut16(vi, space, offset, length, buf16)

viMoveOut32(vi, space, offset, length, buf32)

viMoveOut64(vi, space, offset, length, buf64)

viMoveIn8Ex(vi, space, offset64, length, buf8)

viMoveIn16Ex(vi, space, offset64, length, buf16)

viMoveIn32Ex(vi, space, offset64, length, buf32)

viMoveIn64Ex(vi, space, offset64, length, buf64)

viMoveOut8Ex(vi, space, offset64, length, buf8)

viMoveOut16Ex(vi, space, offset64, length, buf16)

viMoveOut32Ex(vi, space, offset64, length, buf32)

viMoveOut64Ex(vi, space, offset64, length, buf64)

viMove(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

viMoveAsync(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

viMoveEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length)

viMoveAsyncEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId)

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

viMapAddressEx(vi, mapSpace, mapBase64, mapSize, access, suggested, address)

viUnmapAddress(vi)

viPeek8(vi, addr, val8)

viPeek16(vi, addr, val16)

viPeek32(vi, addr, val32)

viPeek64(vi, addr, val64)

viPoke8(vi, addr, val8)

viPoke16(vi, addr, val16)

viPoke32(vi, addr, val32)

viPoke64(vi, addr, val64)

viMemAlloc(vi, size, offset)

viMemFree(vi, offset)

viMemAllocEx(vi, size, offset64)

viMemFreeEx(vi, offset64)

**RULE 5.2.7**

All MEMACC resource implementations **SHALL** support the operations viIn8(), viIn16(), viIn32(), viIn64(), viOut8(), viOut16(), viOut32(), viOut64(), viMoveIn8(), viMoveIn16(), viMoveIn32(), viMoveIn64(), viMoveOut8(), viMoveOut16(), viMoveOut32(), viMoveOut64(), viMoveIn8Ex(), viMoveIn16Ex(), viMoveIn32Ex(), viMoveIn64Ex(), viMoveOut8Ex(), viMoveOut16Ex(), viMoveOut32Ex(), viMoveOut64Ex(), viMove(), viMoveAsync(), viMoveEx(), viMoveAsync(), viMapAddress(), viMapAddressEx(), viUnmapAddress(), viPeek8(), viPeek16(), viPeek32(), viPeek64(), viPoke8(), viPoke16(),viPoke32(), and viPoke64().

**RULE 5.2.8**

A MEMACC resource implementation for a PXI system **SHALL** support the operations viMemAlloc(), viMemFree(), viMemAllocEx(), and viMemFreeEx().

5.3 GPIB Bus Interface Resource

This section describes the resource that is provided to encapsulate the operations and properties of a raw GPIB interface (reading, writing, triggering, and so on). A VISA GPIB Bus Interface (INTFC) Resource, like any other resource, defines the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

5.3.1 INTFC Resource Overview

The INTFC Resource lets a controller interact with any devices connected to the board associated with this resource. Services are provided to send blocks of data onto the bus, request blocks of data from the bus, trigger devices on the bus, and send miscellaneous commands to any or all devices. In addition, the controller can directly query and manipulate specific lines on the bus, and also pass control to other devices with controller capability. These services are described in detail in the remainder of this section. The Basic I/O and Formatted I/O services are also described in the INSTR Resource Overview in section 5.1.1.

5.3.2 INTFC Resource Attributes

**Generic INTFC Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | ViUInt16 | VI\_INTF\_GPIB |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_SEND\_END\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TERMCHAR | R/W | Local | ViUInt8 | 0 to FFh |
| VI\_ATTR\_TERMCHAR\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_DEV\_STATUS\_BYTE | RW | Global | ViUInt8 | 0 to FFh |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | RW | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | RW | Local | ViBoolean | VI\_TRUE VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |
| VI\_ATTR\_WR\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |

# GPIB Specific INTFC Resource Attributes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | | **Range** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | RW | Global | | ViUInt16 | 0 to 30 |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | RW | Global | | ViUInt16 | 0 to 31, VI\_NO\_SEC\_ADDR |
| VI\_ATTR\_GPIB\_REN\_STATE | RO | Global | | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_GPIB\_ATN\_STATE | RO | Global | | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |

(continues)

# GPIB Specific INTFC Resource Attributes (Continued)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | | **Range** |
| VI\_ATTR\_GPIB\_NDAC\_STATE | RO | Global | | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_GPIB\_SRQ\_STATE | RO | Global | | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_GPIB\_CIC\_STATE | RO | Global | | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE | RW | Global | | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_GPIB\_HS488\_CBL\_LEN | RW | Global | | ViInt16 | 1 to 15, VI\_GPIB\_HS488\_DISABLED, VI\_GPIB\_HS488\_NIMPL |
| VI\_ATTR\_GPIB\_ADDR\_STATE | RO | Global | | ViInt16 | VI\_GPIB\_UNADDRESSED VI\_GPIB\_TALKER VI\_GPIB\_LISTENER |

**Generic INTFC Resource Attributes**

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_SEND\_END\_EN Whether to assert END during the transfer of the last byte of the buffer.

VI\_ATTR\_TERMCHAR Termination character. When the termination character is read and VI\_ATTR\_TERMCHAR\_EN is enabled during a read operation, the read operation terminates.

VI\_ATTR\_TERMCHAR\_EN Flag that determines whether the read operation should terminate when a termination character is received.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_DEV\_STATUS\_BYTE This attribute specifies the 488-style status byte of the local controller associated with this session.

If this attribute is written and bit 6 (0x40) is set, this device or controller will assert a service request (SRQ) if it is defined for this interface.

VI\_ATTR\_WR\_BUF\_OPER\_MODE Determines the operational mode of the write buffer. When the operational mode is set to VI\_FLUSH\_WHEN\_FULL (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up.

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the write buffer is flushed under the same conditions, and also every time a viPrintf() operation completes.

VI\_ATTR\_DMA\_ALLOW\_EN This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VI\_ATTR\_RD\_BUF\_OPER\_MODE Determines the operational mode of the read buffer. When the operational mode is set to VI\_FLUSH\_DISABLE (default), the buffer is flushed only on explicit calls to viFlush().

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the buffer is flushed every time a viScanf() operation completes.

VI\_ATTR\_FILE\_APPEND\_EN This attribute specifies whether viReadToFile() will overwrite (truncate) or append when opening a file.

**GPIB Specific INTFC Attributes**

VI\_ATTR\_GPIB\_PRIMARY\_ADDR Primary address of the local GPIB controller used by the given session.

VI\_ATTR\_GPIB\_SECONDARY\_ADDR Secondary address of the local GPIB controller used by the given session.

VI\_ATTR\_GPIB\_REN\_STATE This attribute returns the current state of the GPIB REN (Remote ENable) interface line.

VI\_ATTR\_GPIB\_ATN\_STATE This attribute shows the current state of the GPIB ATN (ATtentioN) interface line.

VI\_ATTR\_GPIB\_NDAC\_STATE This attribute shows the current state of the GPIB NDAC (Not Data ACcepted) interface line.

VI\_ATTR\_GPIB\_SRQ\_STATE This attribute shows the current state of the GPIB SRQ (Service ReQuest) interface line.

VI\_ATTR\_GPIB\_CIC\_STATE This attribute shows whether the specified GPIB interface is currently CIC (controller in charge).

VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE This attribute shows whether the specified GPIB interface is currently the system controller. In some implementations, this attribute may be modified only through a configuration utility. On these systems, this attribute is read only (RO).

VI\_ATTR\_GPIB\_HS488\_CBL\_LEN This attribute specifies the total number of meters of GPIB cable used in the specified GPIB interface. If HS488 is not implemented, querying this attribute should return the value VI\_GPIB\_HS488\_NIMPL. On these systems, trying to set this attribute value will return the error VI\_ERROR\_NSUP\_ATTR\_STATE.

VI\_ATTR\_GPIB\_ADDR\_STATE This attribute shows whether the specified GPIB interface is currently addressed to talk or listen, or is not addressed.

**RULE 5.3.1**

All INTFC resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_NUM, VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_SEND\_END\_EN, VI\_ATTR\_TERMCHAR, VI\_ATTR\_TERMCHAR\_EN, VI\_ATTR\_TMO\_VALUE, VI\_ATTR\_DEV\_STATUS\_BYTE, VI\_ATTR\_WR\_BUF\_OPER\_MODE, VI\_ATTR\_DMA\_ALLOW\_EN, VI\_ATTR\_RD\_BUF\_OPER\_MODE, and VI\_ATTR\_FILE\_APPEND\_EN.

**RULE 5.3.2**

An INTFC resource implementation for a GPIB system **SHALL** support the attributes VI\_ATTR\_GPIB\_PRIMARY\_ADDR, VI\_ATTR\_GPIB\_SECONDARY\_ADDR, VI\_ATTR\_GPIB\_REN\_STATE, VI\_ATTR\_GPIB\_ATN\_STATE, VI\_ATTR\_GPIB\_NDAC\_STATE, VI\_ATTR\_GPIB\_SRQ\_STATE, VI\_ATTR\_GPIB\_CIC\_STATE, VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE, VI\_ATTR\_GPIB\_HS488\_CBL\_LEN, and VI\_ATTR\_GPIB\_ADDR\_STATE.

5.3.3 INTFC Resource Events

**VI\_EVENT\_GPIB\_CIC**

**Description**

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

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_GPIB\_CIC |
| VI\_ATTR\_GPIB\_RECV\_CIC\_STATE | RO | ViBoolean | VI\_TRUE VI\_FALSE |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_GPIB\_TALK**

**Description**

Notification that the GPIB controller has been addressed to talk.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_GPIB\_TALK |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_GPIB\_LISTEN**

**Description**

Notification that the GPIB controller has been addressed to listen.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_GPIB\_LISTEN |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_CLEAR**

**Description**

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

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_CLEAR |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_TRIG**

**Description**

Notification that a trigger interrupt was received from the interface.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_TRIG |
| VI\_ATTR\_RECV\_TRIG\_ID | RO | ViInt16 | VI\_TRIG\_SW |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_RECV\_TRIG\_ID The identifier of the triggering mechanism on which the specified trigger event was received.

**VI\_EVENT\_IO\_COMPLETION**

**Description**

Notification that an asynchronous operation has completed.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_IO\_COMPLETION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_JOB\_ID | RO | ViJobId | N/A |
| VI\_ATTR\_BUFFER | RO | ViBuf | N/A |
| VI\_ATTR\_RET\_COUNT | RO | ViBusSize | \* |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |
| VI\_ATTR\_RET\_COUNT\_32 | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RET\_COUNT\_64\*\* | RO | ViUInt64 | 0 to FFFFFFFFFFFFFFFFh |

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

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

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS This field contains the return code of the asynchronous I/O operation that has completed.

VI\_ATTR\_JOB\_ID This field contains the job ID of the asynchronous operation that has completed.

VI\_ATTR\_BUFFER This field contains the address of a buffer that was used in an asynchronous operation.

VI\_ATTR\_RET\_COUNT This field contains the actual number of elements that were

VI\_ATTR\_RET\_COUNT\_32 asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_64

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

For more information on VI\_ATTR\_OPER\_NAME, see its definition in Section 3.7.2.3, *VI\_EVENT\_EXCEPTION*.

**RULE 5.3.3**

All INTFC resource implementations **SHALL** support the generation of the events VI\_EVENT\_GPIB\_CIC, VI\_EVENT\_GPIB\_TALK, VI\_EVENT\_GPIB\_LISTEN, VI\_EVENT\_CLEAR, VI\_EVENT\_TRIG, VI\_EVENT\_SERVICE\_REQ, and VI\_EVENT\_IO\_COMPLETION.

5.3.4 INTFC Resource Operations

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, fileName, count, retCount)

viAssertTrigger(vi, protocol)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viBufRead(vi, buf, count, retCount)

viGpibControlREN(vi, mode)

viGpibControlATN (vi, mode)

viGpibPassControl(vi, primAddr, secAddr)

viGpibCommand(vi, buf, count, retCount)

viGpibSendIFC(vi)

**RULE 5.3.4**

All INTFC resource implementations **SHALL** support the operations viRead(), viReadAsync(), viReadToFile(), viWrite(), viWriteAsync(), viWriteFromFile(), viAssertTrigger(), viSetBuf(), viFlush(), viPrintf(), viVPrintf(), viSPrintf(), viVSPrintf(), viBufWrite(), viScanf(), viVScanf(), viSScanf(), viVSScanf(), viBufRead(), viGpibControlREN(), viGpibControlATN(), viGpibPassControl(), viGpibCommand(), and viGpibSendIFC().

5.4 Mainframe Backplane Resource

The Mainframe Backplane (BACKPLANE) Resource encapsulates the VXI-defined and PXI-defined operations and properties of the backplane in a VXIbus or PXI system. A VISA Mainframe Backplane Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

5.4.1 BACKPLANE Resource Overview

The BACKPLANE Resource lets a controller query and manipulate specific lines on a specific mainframe in a given VXI or PXI system. Services are provided to map, unmap, assert, and receive hardware triggers, and also to assert various utility and interrupt signals. This includes advanced functionality that may not be available in all implementations or all vendors’ controllers. These services are described in detail in the remainder of this section.

A VXI system with an embedded CPU with one mainframe will always have exactly one BACKPLANE resource. Valid examples of resource strings for this are VXI0::0::BACKPLANE and VXI::BACKPLANE. A multi-chassis VXI system may provide only one BACKPLANE resource total, but the recommended way is to provide one BACKPLANE resource per chassis, with the resource string address corresponding to the attribute VI\_ATTR\_MAINFRAME\_LA. If a multi-chassis VXI system provides only one BACKPLANE resource, it is assumed to control the backplane resources in all chassis.

A PXI system will contain one BACKPLANE resource for each configured chassis, with the resource string address corresponding to the attribute VI\_ATTR\_PXI\_CHASSIS.

**RULE 5.4.1**

A VXI or GPIB-VXI implementation that supports the BACKPLANE resource **SHALL** provide at least one BACKPLANE resource per VXI or GPIB-VXI system.

**RECOMMENDATION 5.4.1**

A VXI or GPIB-VXI implementation should provide one BACKPLANE resource per VXI chassis.

**OBSERVATION 5.4.1**

Some VXI or GPIB-VXI implementations view all chassis in a VXI system as one entity. In these configurations, separate BACKPLANE resources are not possible.

**RULE 5.4.2**

A PXI implementation **SHALL** provide one BACKPLANE resource per configured PXI chassis.

5.4.2 BACKPLANE Resource Attributes

**Generic BACKPLANE Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | ViUInt16 | VI\_INTF\_VXI  VI\_INTF\_GPIB\_VXI  VI\_INTF\_PXI |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_TRIG\_ID | R/W\* | Local | ViInt16 | VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_SLOT1 to  VI\_TRIG\_STAR\_SLOT12;  VI\_TRIG\_STAR\_VXI0 to  VI\_TRIG\_STAR\_VXI2 |

**VXI and GPIB-VXI Specific BACKPLANE Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
|  |  |  |  |  |
| VI\_ATTR\_MAINFRAME\_LA | RO | Global | ViInt16 | 0 to 255 VI\_UNKNOWN\_LA |
| VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_VXI\_VME\_INTR\_STATUS | RO | Global | ViUInt16 | N/A |
| VI\_ATTR\_VXI\_TRIG\_STATUS | RO | Global | ViUInt32 | N/A |
| VI\_ATTR\_VXI\_TRIG\_SUPPORT | RO | Global | ViUInt32 | N/A |

**PXI Specific BACKPLANE Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_MANF\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_MODEL\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_PXI\_CHASSIS | RO | Global | ViInt16 | 1 to 32767 |
| VI\_ATTR\_PXI\_TRIG\_BUS | RW | Local | ViInt16 | -1, 1 to 3 |
| VI\_ATTR\_PXI\_SRC\_TRIG\_BUS | RW | Local | ViInt16 | -1, 1 to 3 |
| VI\_ATTR\_PXI\_DEST\_TRIG\_BUS | RW | Local | ViInt16 | -1, 1 to 3 |

**Generic BACKPLANE Resource Attributes**

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_TRIG\_ID Identifier for the current triggering mechanism.

**VXI and GPIB-VXI Specific BACKPLANE Resource Attributes**

VI\_ATTR\_MAINFRAME\_LA This is the logical address of a given device in the mainframe, usually the device with the lowest logical address. Other possible values include the logical address of the slot-0 controller or of the parent-side extender. Often, these are all the same value. The purpose of this attribute is to provide a unique ID for each mainframe. A VISA manufacturer can choose any of these values, but must be consistent across mainframes. If this value is not known, the attribute value returned is VI\_UNKNOWN\_LA.

VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE This attribute shows the current state of the VXI/VME SYSFAIL (SYStem FAILure) backplane line.

VI\_ATTR\_VXI\_VME\_INTR\_STATUS This attribute shows the current state of the VXI/VME interrupt lines. This is a bit vector with bits 0-6 corresponding to interrupt lines 1-7.

VI\_ATTR\_VXI\_TRIG\_STATUS This attribute shows the current state of the VXI trigger lines. This is a bit vector. Bits 0-7 correspond to VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7. Bits 8-13 correspond to VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5. Bits 14-25 correspond to VI\_TRIG\_STAR\_SLOT1 to VI\_TRIG\_STAR\_SLOT12. Bit 27 corresponds to VI\_TRIG\_PANEL\_IN and bit 28 corresponds to VI\_TRIG\_PANEL\_OUT. Bits 29-31 correspond to VI\_TRIG\_STAR\_VXI0 to VI\_TRIG\_STAR\_VXI2.

VI\_ATTR\_VXI\_TRIG\_SUPPORT This attribute shows which VXI trigger lines this implementation supports. This is a bit vector. Bits 0-7 correspond to VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7. Bits 8-13 correspond to VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5. Bits 14-25 correspond to VI\_TRIG\_STAR\_SLOT1 to VI\_TRIG\_STAR\_SLOT12. Bit 27 corresponds to VI\_TRIG\_PANEL\_IN and bit 28 corresponds to VI\_TRIG\_PANEL\_OUT. Bits 29-31 correspond to VI\_TRIG\_STAR\_VXI0 to VI\_TRIG\_STAR\_VXI2.

**PXI Specific BACKPLANE Resource Attributes**

VI\_ATTR\_MANF\_NAME This string attribute is the chassis manufacturer name.

VI\_ATTR\_MODEL\_NAME This string attribute is the model name of the chassis.

VI\_ATTR\_PXI\_CHASSIS Specifies the PXI chassis number of this resource.

VI\_ATTR\_PXI\_TRIG\_BUS Specifies the segment to use in calls to viAssertTrigger.

VI\_ATTR\_PXI\_SRC\_TRIG\_BUS Specifies the segment to use to qualify trigSrc in calls to viMapTrigger.

VI\_ATTR\_PXI\_DEST\_TRIG\_BUS Specifies the segment to use to qualify trigDest in calls to viMapTrigger.

**RULE 5.4.3**

All BACKPLANE resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_NUM, VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_TRIG\_ID , and VI\_ATTR\_TMO\_VALUE.

**RULE 5.4.4**

A BACKPLANE resource implementation for a VXI or GPIB-VXI system **SHALL** support the attributes, VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE, VI\_ATTR\_VXI\_VME\_INTR\_STATUS, VI\_ATTR\_VXI\_TRIG\_STATUS, VI\_ATTR\_MAINFRAME\_LA, and VI\_ATTR\_VXI\_TRIG\_SUPPORT.

**RULE 5.4.5**

A BACKPLANE resource implementation for a PXI system **SHALL** support the attributes, VI\_ATTR\_MANF\_NAME, VI\_ATTR\_MODEL\_NAME, VI\_ATTR\_PXI\_CHASSIS, VI\_ATTR\_PXI\_TRIG\_BUS, VI\_ATTR\_PXI\_SRC\_TRIG\_BUS, and VI\_ATTR\_PXI\_DEST\_TRIG\_BUS.

**RULE 5.4.6**

A BACKPLANE resource implementation for a PXI system **SHALL** use the Trigger Manager interface for the backplane as defined in the PXI-9 specification for reserving and mapping trigger resources.

**RULE 5.4.7**

A BACKPLANE resource implementation for a PXI system **SHALL** read pxisys.ini and pxiesys.ini to detect trigger bus resources.

5.4.3 BACKPLANE Resource Events

**VI\_EVENT\_TRIG**

**Description**

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

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_TRIG |
| VI\_ATTR\_RECV\_TRIG\_ID | RO | ViInt16 | VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_SLOT1 to  VI\_TRIG\_STAR\_SLOT12 |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_RECV\_TRIG\_ID The identifier of the triggering mechanism on which the specified trigger event was received.

**VI\_EVENT\_VXI\_VME\_SYSFAIL**

**Description**

Notification that the VXI/VME SYSFAIL\* line has been asserted.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_VXI\_VME\_SYSFAIL |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_VXI\_VME\_SYSRESET**

**Description**

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

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_VXI\_VME\_SYSRESET |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**RULE 5.4.8**

A BACKPLANE resource implementation for a VXI system **SHALL** support the generation of the events

VI\_EVENT\_VXI\_VME\_SYSFAIL, VI\_EVENT\_VXI\_VME\_SYSRESET, and VI\_EVENT\_TRIG.

5.4.4 BACKPLANE Resource Operations

viAssertTrigger(vi, protocol)

viAssertUtilSignal(vi, line)

viAssertIntrSignal(vi, mode, statusID)

viMapTrigger(vi, trigSrc, trigDest, mode)

viUnmapTrigger(vi, trigSrc, trigDest)

viPxiReserveTriggers(vi, cnt, trigBuses, trigLines, failureIndex)

**RULE 5.4.9**

All VXI and GPIB-VXI BACKPLANE resource implementations **SHALL** support the operations viAssertTrigger(), viAssertUtilSignal(), viAssertIntrSignal(), viMapTrigger(), viUnmapTrigger().

**RULE 5.4.10**

All PXI BACKPLANE resource implementations **SHALL** support the operations viAssertTrigger(), viMapTrigger(), viUnmapTrigger(), and viPxiReserveTriggers().

5.5 Servant Device-Side Resource

The Servant (SERVANT) Resource encapsulates the operations and properties of the capabilities of a device and a device’s view of the system in which it exists. A VISA Servant Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

5.5.1 SERVANT Resource Overview

The SERVANT Resource exposes the device-side functionality of the device associated with this resource. Services are provided to receive blocks of data from a commander and respond with blocks of data in return, setting a 488-style status byte, and receiving device clear and trigger events. These services are described in detail in the remainder of this section. The Basic I/O and Formatted I/O services are also described in the INSTR Resource Overview in section 5.1.1.

The SERVANT resource is a class for advanced users who want to write firmware code that exports device functionality across multiple interfaces. Most VISA users will not need this level of functionality and should not use the SERVANT resource in their applications.

A VISA user of the TCPIP SERVANT resource should be aware that each VISA session corresponds to a unique socket connection. If the user opens only one SERVANT session, this precludes multiple clients from accessing the device.

5.5.2 SERVANT Resource Attributes

## Generic SERVANT Resource Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | ViUInt16 | VI\_INTF\_VXI  VI\_INTF\_GPIB  VI\_INTF\_TCPIP |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_SEND\_END\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TERMCHAR | R/W | Local | ViUInt8 | 0 to FFh |
| VI\_ATTR\_TERMCHAR\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_DEV\_STATUS\_BYTE | RW | Global | ViUInt8 | 0 to FFh |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | RW | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | RW | Local | ViBoolean | VI\_TRUE VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |
| VI\_ATTR\_WR\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |

## GPIB Specific SERVANT Resource Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | R/W | Global | ViUInt16 | 0 to 30 |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | R/W | Global | ViUInt16 | 0 to 31, VI\_NO\_SEC\_ADDR |
| VI\_ATTR\_GPIB\_REN\_STATE | RO | Global | ViInt16 | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN |
| VI\_ATTR\_GPIB\_ADDR\_STATE | RO | Global | ViInt16 | VI\_GPIB\_UNADDRESSED VI\_GPIP\_TALKER VI\_GPIB\_LISTENER |

## VXI Specific SERVANT Resource Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_VXI\_LA | RO | Global | ViInt16 | 0 to 511 |
| VI\_ATTR\_CMDR\_LA | RO | Global | ViInt16 | 0 to 255  VI\_UNKNOWN\_LA |

**TCPIP Specific SERVANT Resource Attributes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_TCPIP\_DEVICE\_NAME | RO | Global | ViString | N/A |

**Generic SERVANT Resource Attributes**

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_SEND\_END\_EN Whether to assert END during the transfer of the last byte of the buffer.

VI\_ATTR\_TERMCHAR Termination character. When the termination character is read and VI\_ATTR\_TERMCHAR\_EN is enabled during a read operation, the read operation terminates.

VI\_ATTR\_TERMCHAR\_EN Flag that determines whether the read operation should terminate when a termination character is received.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_DEV\_STATUS\_BYTE This attribute specifies the 488-style status byte of the local controller associated with this session.

VI\_ATTR\_WR\_BUF\_OPER\_MODE Determines the operational mode of the write buffer. When the operational mode is set to VI\_FLUSH\_WHEN\_FULL (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up.

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the write buffer is flushed under the same conditions, and also every time a viPrintf() operation completes.

VI\_ATTR\_DMA\_ALLOW\_EN This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VI\_ATTR\_RD\_BUF\_OPER\_MODE Determines the operational mode of the read buffer. When the operational mode is set to VI\_FLUSH\_DISABLE (default), the buffer is flushed only on explicit calls to viFlush().

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the buffer is flushed every time a viScanf() operation completes.

VI\_ATTR\_FILE\_APPEND\_EN This attribute specifies whether viReadToFile() will overwrite (truncate) or append when opening a file.

**GPIB Specific SERVANT Resource Attributes**

VI\_ATTR\_GPIB\_PRIMARY\_ADDR Primary address of the local GPIB controller used by the given session.

VI\_ATTR\_GPIB\_SECONDARY\_ADDR Secondary address of the local GPIB controller used by the given session.

VI\_ATTR\_GPIB\_REN\_STATE This attribute returns the current state of the GPIB REN (Remote ENable) interface line.

VI\_ATTR\_GPIB\_ADDR\_STATE This attribute showswhether the specified GPIB interface is currently addressed to talk to listen, or to not addressed.

**VXI Specific SERVANT Resource Attributes**

VI\_ATTR\_VXI\_LA Logical address of the VXI or VME device used by the given session. For a VME device, the logical address is actually a pseudo-address in the range 256 to 511.

VI\_ATTR\_CMDR\_LA Logical address of the commander of the VXI device used by the given session.

VI\_ATTR\_TCPIP\_DEVICE\_NAME This specifies the LAN device name used by the VXI-11 protocol during connection.

**RULE 5.5.1**

All SERVANT resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_NUM, VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_SEND\_END\_EN, VI\_ATTR\_TERMCHAR, VI\_ATTR\_TERMCHAR\_EN, VI\_ATTR\_TMO\_VALUE, VI\_ATTR\_WR\_BUF\_OPER\_MODE, VI\_ATTR\_RD\_BUF\_OPER\_MODE, VI\_ATTR\_DEV\_STATUS\_BYTE, VI\_ATTR\_DMA\_ALLOW\_EN, and VI\_ATTR\_FILE\_APPEND\_EN.

**RULE 5.5.2**

A SERVANT resource implementation for a GPIB system **SHALL** support the attributes VI\_ATTR\_GPIB\_PRIMARY\_ADDR, VI\_ATTR\_GPIB\_SECONDARY\_ADDR, VI\_ATTR\_GPIB\_REN\_STATE, and VI\_ATTR\_GPIB\_ADDR\_STATE**.**

**RULE 5.5.3**

A SERVANT resource implementation for a VXI system **SHALL** support the attributes VI\_ATTR\_VXI\_LA and VI\_ATTR\_CMDR\_LA**.**

**RULE 5.5.4**

**IF** a SERVANT resource implementation does not support DMA transfers, **AND** the attribute is VI\_ATTR\_DMA\_ALLOW\_EN, **AND** the attribute state is VI\_TRUE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE.

5.5.3 SERVANT Resource Events

**VI\_EVENT\_CLEAR**

**Description**

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

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_CLEAR |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_IO\_COMPLETION**

**Description**

Notification that an asynchronous operation has completed.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_IO\_COMPLETION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_JOB\_ID | RO | ViJobId | N/A |
| VI\_ATTR\_BUFFER | RO | ViBuf | N/A |
| VI\_ATTR\_RET\_COUNT | RO | ViBusSize | \* |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |
| VI\_ATTR\_RET\_COUNT\_32 | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RET\_COUNT\_64\*\* | RO | ViUInt64 | 0 to FFFFFFFFFFFFFFFFh |

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

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

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS This field contains the return code of the asynchronous I/O operation that has completed.

VI\_ATTR\_JOB\_ID This field contains the job ID of the asynchronous operation that has completed.

VI\_ATTR\_BUFFER This field contains the address of a buffer that was used in an asynchronous operation.

VI\_ATTR\_RET\_COUNT This field contains the actual number of elements that were

VI\_ATTR\_RET\_COUNT\_32 asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_64

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

For more information on VI\_ATTR\_OPER\_NAME, see its definition in Section 3.7.2.3, *VI\_EVENT\_EXCEPTION*.

**VI\_EVENT\_GPIB\_TALK**

**Description**

Notification that the GPIB controller has been addressed to talk.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_GPIB\_TALK |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_GPIB\_LISTEN**

**Description**

Notification that the GPIB controller has been addressed to listen.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_GPIB\_LISTEN |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_TRIG**

**Description**

Notification that the local controller has been triggered.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_TRIG |
| VI\_ATTR\_RECV\_TRIG\_ID | RO | ViInt16 | VI\_TRIG\_SW |

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_RECV\_TRIG\_ID The identifier of the triggering mechanism on which the specified trigger event was received.

**VI\_EVENT\_VXI\_VME\_SYSRESET**

**Description**

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

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_VXI\_VME\_SYSRESET |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

**VI\_EVENT\_TCPIP\_CONNECT**

**Description**

Notification that a TCP/IP connection has been made.

**Event Attribute**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_TCPIP\_CONNECT |
| VI\_ATTR\_RECV\_TCPIP\_ADDR | RO | ViString | N/A |

**Event Attribute Description**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_RECV\_TCPIP\_ADDR This is the TCP/IP address of the device from which the session received a connection.

**RULE 5.5.5**

All SERVANT resource implementations **SHALL** support the events VI\_EVENT\_IO\_COMPLETION, VI\_EVENT\_TRIG, and VI\_EVENT\_CLEAR.

**RULE 5.5.6**

A SERVANT resource implementation for a GPIB system **SHALL** support the events VI\_EVENT\_GPIB\_TALK and VI\_EVENT\_GPIB\_LISTEN.

**RULE 5.5.7**

A SERVANT resource implementation for a VXI system **SHALL** support the event VI\_EVENT\_VXI\_VME\_SYSRESET.

**RULE 5.5.8**

A SERVANT resource implementation for a TCPIP system **SHALL** support the event   
VI\_EVENT\_TCPIP\_CONNECT.

5.5.4 SERVANT Resource Operations

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, fileName, count, retCount)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viBufRead(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

viAssertIntrSignal(vi, mode, statusID)

viAssertUtilSignal(vi, line)

**RULE 5.5.9**

All SERVANT resource implementations **SHALL** support the operations viRead(), viReadAsync(), viWrite(), viWriteAsync(), viSetBuf(), viBufRead(), viScanf(), viPrintf(), viVPrintf(), viFlush(),viBufWrite(), viSScanf(), viVSScanf(), viSPrintf(), viVSPrintf(), viReadToFile(), and viWriteFromFile().

**RULE 5.5.10**

A SERVANT resource implementation for a VXI system **SHALL** support the operations viAssertIntrSignal and viAssertUtilSignal().

**RULE 5.5.11**

A SERVANT resource implementation for a TCPIP system **SHALL** use the VXI-11 protocol.

5.6 TCP/IP Socket Resource

The TCP/IP Socket (SOCKET) Resource encapsulates the operations and properties of the capabilities of a raw network socket connection using TCP/IP. A VISA Socket Resource, like any other resource, starts with the basic operations and attributes of the VISA Resource Template. For example, modifying the state of an attribute is done via the operation viSetAttribute(), which is defined in the VISA Resource Template. Although the following resource does not have viSetAttribute() listed in its operations, it provides the operation because it is defined in the VISA Resource Template. From this basic set, each resource adds its specific operations and attributes that allow it to perform its dedicated task.

5.6.1 SOCKET Resource Overview

The SOCKET Resource exposes the capability of a raw network socket connection over TCP/IP. This ususally means Ethernet but the protocol is not restricted to that physical interface. Services are provided to send and receive blocks of data. If the device is capable of communicating with 488.2-style strings, an attribute setting also allows sending software triggers, querying a 488-style status byte, and sending a device clear message. These services are described in detail in the remainder of this section. The Basic I/O and Formatted I/O services are also described in the INSTR Resource Overview in section 5.1.1.

5.6.2 SOCKET Resource Attributes

## Generic SOCKET Resource Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_INTF\_NUM | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_INTF\_TYPE | RO | Global | ViUInt16 | VI\_INTF\_TCPIP |
| VI\_ATTR\_INTF\_INST\_NAME | RO | Global | ViString | N/A |
| VI\_ATTR\_SEND\_END\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TERMCHAR | R/W | Local | ViUInt8 | 0 to FFh |
| VI\_ATTR\_TERMCHAR\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | R/W | Local | ViUInt32 | VI\_TMO\_IMMEDIATE  1 to FFFFFFFEh  VI\_TMO\_INFINITE |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | R/W | Local | ViBoolean | VI\_TRUE  VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | R/W | Local | ViUInt16 | VI\_FLUSH\_ON\_ACCESS  VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | R/W | Local | ViBoolean | VI\_TRUE VI\_FALSE |
| VI\_ATTR\_IO\_PROT | R/W | Local | ViUInt16 | VI\_PROT\_NORMAL VI\_PROT\_4882\_STRS |
| VI\_ATTR\_RD\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |
| VI\_ATTR\_WR\_BUF\_SIZE | RO | Local | ViUInt32 | N/A |

## TCPIP Specific SOCKET Resource Attributes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | | **Data Type** | **Range** |
| VI\_ATTR\_TCPIP\_ADDR | RO | Global | ViString | N/A |
| VI\_ATTR\_TCPIP\_HOSTNAME | RO | Global | ViString | N/A |
| VI\_ATTR\_TCPIP\_PORT | RO | Global | ViUInt16 | 0 to FFFFh |
| VI\_ATTR\_TCPIP\_NODELAY | R/W | Local | ViBoolean | VI\_TRUE, VI\_FALSE |
| VI\_ATTR\_TCPIP\_KEEPALIVE | R/W | Local | ViBoolean | VI\_TRUE, VI\_FALSE |

**Generic SERVANT Resource Attributes**

VI\_ATTR\_INTF\_NUM Board number for the given interface.

VI\_ATTR\_INTF\_TYPE Interface type of the given session.

VI\_ATTR\_INTF\_INST\_NAME Human-readable text describing the given interface.

VI\_ATTR\_SEND\_END\_EN Whether to assert END during the transfer of the last byte of the buffer.

VI\_ATTR\_TERMCHAR Termination character. When the termination character is read and VI\_ATTR\_TERMCHAR\_EN is enabled during a read operation, the read operation terminates.

VI\_ATTR\_TERMCHAR\_EN Flag that determines whether the read operation should terminate when a termination character is received.

VI\_ATTR\_TMO\_VALUE Minimum timeout value to use, in milliseconds. A timeout value of VI\_TMO\_IMMEDIATE means that operations should never wait for the device to respond. A timeout value of VI\_TMO\_INFINITE disables the timeout mechanism.

VI\_ATTR\_WR\_BUF\_OPER\_MODE Determines the operational mode of the write buffer. When the operational mode is set to VI\_FLUSH\_WHEN\_FULL (default), the buffer is flushed when an END indicator is written to the buffer, or when the buffer fills up.

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the write buffer is flushed under the same conditions, and also every time a viPrintf() operation completes.

VI\_ATTR\_DMA\_ALLOW\_EN This attribute specifies whether I/O accesses should use DMA (VI\_TRUE) or Programmed I/O (VI\_FALSE). In some implementations, this attribute may have global effects even though it is documented to be a local attribute. Since this affects performance and not functionality, that behavior is acceptable.

VI\_ATTR\_RD\_BUF\_OPER\_MODE Determines the operational mode of the read buffer. When the operational mode is set to VI\_FLUSH\_DISABLE (default), the buffer is flushed only on explicit calls to viFlush().

If the operational mode is set to VI\_FLUSH\_ON\_ACCESS, the buffer is flushed every time a viScanf() operation completes.

VI\_ATTR\_FILE\_APPEND\_EN This attribute specifies whether viReadToFile() will overwrite (truncate) or append when opening a file.

VI\_ATTR\_IO\_PROT Specifies which protocol to use.

**TCPIP Specific SOCKET Resource Attributes**

VI\_ATTR\_TCPIP\_ADDR This is the TCPIP address of the device to which the session is connected. This string is formatted in dot notation.

VI\_ATTR\_TCPIP\_HOSTNAME This specifies the host name of the device. If no host name is available, this attribute returns an empty string.

VI\_ATTR\_TCPIP\_PORT This specifies the port number for a given TCPIP address. For a TCPIP SOCKET resource, this is a required part of the address string.

VI\_ATTR\_TCPIP\_NODELAY The Nagle algorithm is disabled when this attribute is enabled (and vice versa). The Nagle algorithm improves network performance by buffering “send” data until a full-size packet can be sent. This attribute is enabled by default in VISA to verify that synchronous writes get flushed immediately.

VI\_ATTR\_TCPIP\_KEEPALIVE An application can request that a TCP/IP provider enable the use of “keep-alive” packets on TCP connections by turning on this attribute. If a connection is dropped as a result of “keep-alives,” the error code VI\_ERROR\_CONN\_LOST is returned to current and subsequent I/O calls on the session.

**RULE 5.6.1**

All SOCKET resource implementations **SHALL** support the attributes VI\_ATTR\_INTF\_NUM, VI\_ATTR\_INTF\_TYPE, VI\_ATTR\_INTF\_INST\_NAME, VI\_ATTR\_SEND\_END\_EN, VI\_ATTR\_TERMCHAR, VI\_ATTR\_TERMCHAR\_EN, VI\_ATTR\_TMO\_VALUE, VI\_ATTR\_WR\_BUF\_OPER\_MODE, VI\_ATTR\_RD\_BUF\_OPER\_MODE, VI\_ATTR\_DMA\_ALLOW\_EN, and VI\_ATTR\_FILE\_APPEND\_EN.

**RULE 5.6.2**

A SOCKET resource implementation for a TCPIP system **SHALL** support the attributes VI\_ATTR\_TCPIP\_ADDR, VI\_ATTR\_TCPIP\_HOSTNAME, VI\_ATTR\_TCPIP\_PORT, VI\_ATTR\_TCPIP\_NODELAY, and VI\_ATTR\_TCPIP\_KEEPALIVE**.**

**RULE 5.6.3**

**IF** a SOCKET resource implementation does not support DMA transfers, **AND** the attribute is VI\_ATTR\_DMA\_ALLOW\_EN, **AND** the attribute state is VI\_TRUE, **THEN** the call to viSetAttribute() **SHALL** return the completion code VI\_WARN\_NSUP\_ATTR\_STATE**.**

**OBSERVATION 5.6.1**

Since most SOCKET implementations use Ethernet, and Ethernet services do not usually support DMA, trying to enable DMA on a SOCKET resource will most likely return VI\_WARN\_NSUP\_ATTR\_STATE.

5.6.3 SOCKET Resource Events

**VI\_EVENT\_IO\_COMPLETION**

**Description**

Notification that an asynchronous operation has completed.

**Event Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbolic Name** | **Access Privilege** | **Data Type** | **Range** |
| VI\_ATTR\_EVENT\_TYPE | RO | ViEventType | VI\_EVENT\_IO\_COMPLETION |
| VI\_ATTR\_STATUS | RO | ViStatus | N/A |
| VI\_ATTR\_JOB\_ID | RO | ViJobId | N/A |
| VI\_ATTR\_BUFFER | RO | ViBuf | N/A |
| VI\_ATTR\_RET\_COUNT | RO | ViBusSize | \* |
| VI\_ATTR\_OPER\_NAME | RO | ViString | N/A |
| VI\_ATTR\_RET\_COUNT\_32 | RO | ViUInt32 | 0 to FFFFFFFFh |
| VI\_ATTR\_RET\_COUNT\_64\*\* | RO | ViUInt64 | 0 to FFFFFFFFFFFFFFFFh |

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

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

**Event Attribute Descriptions**

VI\_ATTR\_EVENT\_TYPE Unique logical identifier of the event.

VI\_ATTR\_STATUS This field contains the return code of the asynchronous I/O operation that has completed.

VI\_ATTR\_JOB\_ID This field contains the job ID of the asynchronous operation that has completed.

VI\_ATTR\_BUFFER This field contains the address of a buffer that was used in an asynchronous operation.

VI\_ATTR\_RET\_COUNT This field contains the actual number of elements that were

VI\_ATTR\_RET\_COUNT\_32 asynchronously transferred.

VI\_ATTR\_RET\_COUNT\_64

VI\_ATTR\_OPER\_NAME The name of the operation generating the event.

For more information on VI\_ATTR\_OPER\_NAME, see its definition in Section 3.7.2.3, *VI\_EVENT\_EXCEPTION*.

**RULE 5.6.4**

All SOCKET resource implementations **SHALL** support the event VI\_EVENT\_IO\_COMPLETION.

5.6.4 SOCKET Resource Operations

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, filename, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, filename, count, retCount)

viAssertTrigger(vi, protocol)

viReadSTB(vi, status)

viClear(vi)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viBufRead(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

**RULE 5.6.5**

All SOCKET resource implementations **SHALL** support the operations viRead(), viReadAsync(), viReadToFile (), viWrite(), viWriteAsync(), viWriteFromFile (), viAssertTrigger(), viReadSTB(), viClear(), viSetBuf(), viFlush(), viBufRead(), viScanf(), viPrintf(), viVPrintf(), viBufWrite(), viSScanf(), viVSScanf(), viSPrintf(), and viVSPrintf().

Section 6 VISA Resource-Specific Operations

This section describes in detail the operations that are specific to the VISA resources listed in the previous sections. Under the *Related Items* section, each operation includes a list of the resources to which it belongs. For operations that apply to more than one resource but have slightly different behavior for different resources, any resource-specific information will be listed separately at the end of each operation.

These operations are grouped by the type of service they provide. The types of services, listed below, have already been introduced in the previous sections.

1. Basic I/O Services
2. Formatted I/O Services
3. Memory I/O Services
4. Shared Memory Services
5. Interface Specific Services

6.1 Basic I/O Services

6.1.1 viRead(vi, buf, count, retCount)

**Purpose**

Read data from device synchronously.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | The operation completed successfully and the END indicator was received (for interfaces that have END indicators). |
| VI\_SUCCESS\_TERM\_CHAR | The specified termination character was read. |
| VI\_SUCCESS\_MAX\_CNT | The number of bytes read is equal to count. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_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. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_OUTP\_PROT\_VIOL | Device reported an output protocol error 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 | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | No listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_ASRL\_PARITY | A parity error occurred during transfer. |
| VI\_ERROR\_ASRL\_FRAMING | A framing error occurred during transfer. |
| VI\_ERROR\_ASRL\_OVERRUN | An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived. |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

The synchronous read operation synchronously transfers data. The data read is to be stored in the buffer represented by buf. This operation returns only when the transfer terminates. Only one synchronous read operation can occur at any one time.

Table 6.1.1 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viWrite().

**Implementation Requirements**

**OBSERVATION** **6.1.1**

A viRead() operation can complete successfully if one or more of the following conditions were met:

a) END indicator received. b) Termination character read. c) Number of bytes read is equal to count.

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

**RULE 6.1.1**

**IF** an END indicator is received, **AND** VI\_ATTR\_SUPPRESS\_END\_EN is VI\_FALSE, **THEN** viRead() **SHALL** return VI\_SUCCESS, regardless of whether the termination character is received or the number of bytes read is equal to count.

**RULE 6.1.2**

**IF** no END indicator is received, **AND** the termination character is read, **AND** VI\_ATTR\_TERMCHAR\_EN is VI\_TRUE, **THEN** viRead() **SHALL** return VI\_SUCCESS\_TERM\_CHAR, regardless of whether the number of bytes read is equal to count.

**RULE 6.1.3**

**IF** no END indicator is received, **AND** no termination character is read, **AND** the number of bytes read is equal to count, **THEN** viRead() **SHALL** return VI\_SUCCESS\_MAX\_CNT.

**OBSERVATION** **6.1.2**

If you pass VI\_NULL as the retCount parameter to the viRead() operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

**RULE 6.1.4**

**IF** VI\_ATTR\_SUPPRESS\_END\_EN is VI\_TRUE, **THEN** viRead() **SHALL NOT** return VI\_SUCCESS.

**RULE 6.1.5**

**IF** VI\_ATTR\_TERMCHAR\_EN is VI\_FALSE, **THEN** viRead() **SHALL NOT** return VI\_SUCCESS\_TERM\_CHAR.

**RULE 6.1.6**

**IF** vi is a session to an ASRL INSTR resource, **AND** VI\_ATTR\_ASRL\_END\_IN is VI\_ASRL\_END\_NONE, **THEN** viRead() **SHALL NOT** return VI\_SUCCESS.

**RULE 6.1.7**

**IF** vi is a session to an ASRL INSTR resource, **AND** VI\_ATTR\_ASRL\_END\_IN is VI\_ASRL\_END\_TERMCHAR, **THEN** viRead() **SHALL** treat the value stored in VI\_ATTR\_TERMCHAR as an END indicator, regardless of the value of VI\_ATTR\_TERMCHAR\_EN.

**OBSERVATION 6.1.3**

RULES 6.1.4 and 6.1.6 state conditions under which viRead() will not terminate because of an END condition. The operation can still complete successfully due to a termination character or reading the maximum number of bytes requested.

**OBSERVATION 6.1.4**

RULE 6.1.5 states a condition under which viRead() will not terminate because of reading a termination character. The operation can still complete successfully due to reading the maximum number of bytes requested.

6.1.2 viReadAsync(vi, buf, count, jobId)

**Purpose**

Read data from device asynchronously.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| buf | OUT | ViBuf | Represents the location of a buffer to receive data from device. |
| count | IN | ViUInt32 | Number of bytes to be read. |
| jobId | OUT | ViJobId | 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Asynchronous read operation successfully queued. |
| VI\_SUCCESS\_SYNC | Read operation performed synchronously. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_QUEUE\_ERROR | Unable to queue read operation. |
| VI\_ERROR\_IN\_PROGRESS | Unable to start a new asynchronous operation while another asynchronous operation is in progress. |

**Description**

The asynchronous read operation asynchronously transfers data. The data read is to be stored in the buffer represented by buf. This operation normally returns before the transfer terminates. An I/O Completion event will be posted when the transfer is actually completed.

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.

Table 6.1.2 Special Values for jobId Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return a job identifier. |

**Related Items**

See the INSTR resource description. Also see viRead(), viTerminate(), viWrite(), and viWriteAsync().

**Implementation Requirements**

**RULE 6.1.8**

**IF** the output parameter jobId is not VI\_NULL, **THEN** the value in jobId **SHALL** be valid before viReadAsync() begins the data transfer.

**OBSERVATION 6.1.5**

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

**OBSERVATION 6.1.6**

If you pass VI\_NULL as the jobId parameter to the viReadAsync() operation, no jobId will be returned. This option may be useful if only one asynchronous operation will be pending at a given time.

**OBSERVATION 6.1.7**

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

**PERMISSION 6.1.1**

The viReadAsync() operation **MAY** be implemented synchronously, which could be done by using the viRead() operation.

**RULE 6.1.9**

**IF** the viReadAsync() operation is implemented synchronously, **AND** a given invocation of the operation is valid, **THEN** the operation **SHALL** return VI\_SUCCESS\_SYNC, **AND** all status information **SHALL** be returned in a VI\_EVENT\_IO\_COMPLETION.

**OBSERVATION 6.1.8**

The intent of PERMISSION 6.1.1 and RULE 6.1.9 is that an application can use the asynchronous operations transparently, even if the low-level driver used for a given VISA implementation supports only synchronous data transfers.

**RULE 6.1.10**

The status codes returned in the VI\_ATTR\_STATUS field of a VI\_EVENT\_IO\_COMPLETION event resulting from a call to viReadAsync() **SHALL** be the same codes as those listed for viRead().

**OBSERVATION 6.1.9**

The status code VI\_ERROR\_RSRC\_LOCKED can be returned either immediately or from the VI\_EVENT\_IO\_COMPLETION event.

**OBSERVATION 6.1.10**

The contents of the output buffer pointed to by buf are not guaranteed to be valid until the VI\_EVENT\_IO\_COMPLETION event occurs.

**RULE 6.1.11**

For each successful call to viReadAsync(), there **SHALL** be one and only one VI\_EVENT\_IO\_COMPLETION event occurrence.

**RULE 6.1.12**

**IF** the jobId parameter returned from viReadAsync() is passed to viTerminate(), **AND** a VI\_EVENT\_IO\_COMPLETION event has not yet occurred for the specified jobId, **THEN** the viTerminate() operation **SHALL** raise a VI\_EVENT\_IO\_COMPLETION event on the given vi, **AND** the VI\_ATTR\_STATUS field of that event **SHALL** be set to VI\_ERROR\_ABORT.

**RULE 6.1.13**

**IF** the output parameter jobId is not VI\_NULL **AND** the return status from viReadAsync() is successful, **THEN** the value in jobId **SHALL NOT** be VI\_NULL.

**OBSERVATION 6.1.11**

The value VI\_NULL is a reserved jobId and has a special meaning in viTerminate().

6.1.3 viReadToFile(vi, fileName, count, retCount)

**Purpose**

Read data synchronously, and store the transferred data in a file.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | The operation completed successfully and the END indicator was received (for interfaces that have END indicators). |
| VI\_SUCCESS\_TERM\_CHAR | The specified termination character was read. |
| VI\_SUCCESS\_MAX\_CNT | The number of bytes read is equal to count. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_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 during transfer. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| 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 | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | No listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_ASRL\_PARITY | A parity error occurred during transfer. |
| VI\_ERROR\_ASRL\_FRAMING | A framing error occurred during transfer. |
| VI\_ERROR\_ASRL\_OVERRUN | An overrun error occurred during transfer. A character was not read from the hardware before the next character arrived. |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |
| VI\_ERROR\_FILE\_ACCESS | An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights. |
| VI\_ERROR\_FILE\_IO | An error occurred while accessing the specified file. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

This read operation synchronously transfers data. The file specified in fileName is opened in binary write-only mode. If the value of VI\_ATTR\_FILE\_APPEND\_EN is VI\_FALSE, any existing contents are destroyed; otherwise, the file contents are preserved. The data read is written to the file. This operation returns only when the transfer terminates.

This operation is useful for storing raw data to be processed later.

Table 6.1.3 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viRead() and viWriteFromFile().

**Implementation Requirements**

**RULE 6.1.14**

The operation viReadToFile() **SHALL** open the file specified by fileName in binary mode.

**OBSERVATION 6.1.12**

If a VISA implementation uses the ANSI C file operations, the mode used by viReadToFile() should be “wb” or “ab” depending on the value of VI\_ATTR\_FILE\_APPEND\_EN.

**RULE 6.1.15**

The operation viReadToFile() **SHALL** return the success codes VI\_SUCCESS, VI\_SUCCESS\_MAX\_CNT, and VI\_SUCCESS\_TERM\_CHAR under the same conditions as viRead().

6.1.4 viWrite(vi, buf, count, retCount)

**Purpose**

Write data to device synchronously.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Transfer completed. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_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\_INV\_SETUP | Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_NCIC | The interface associated with the given vi is not currently the controller in charge. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_NLISTENERS | No Listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

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

Table 6.1.4 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viRead().

**Implementation Requirements**

**OBSERVATION 6.1.13**

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

6.1.5 viWriteAsync(vi, buf, count, jobId)

**Purpose**

Write data to device asynchronously.

**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. |
| jobId | OUT | ViJobId | 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Asynchronous write operation successfully queued. |
| VI\_SUCCESS\_SYNC | Write operation performed synchronously. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_QUEUE\_ERROR | Unable to queue write operation. |
| VI\_ERROR\_IN\_PROGRESS | Unable to start a new asynchronous operation while another asynchronous operation is in progress. |

**Description**

The write operation asynchronously transfers data. The data to be written is in the buffer represented by buf. This operation normally returns before the transfer terminates. An I/O Completion event will be posted when the transfer is actually completed.

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.

Table 6.1.5 Special Values for jobId Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return a job identifier. |

**Related Items**

See the INSTR resource description. Also see viRead(), viTerminate(), viReadAsync(), and viWrite().

**Implementation Requirements**

**RULE 6.1.16**

**IF** the output parameter jobId is not VI\_NULL, **THEN** the value in jobId **SHALL** be valid before viWriteAsync() begins the data transfer.

**OBSERVATION 6.1.14**

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

**OBSERVATION 6.1.15**

If you pass VI\_NULL as the jobId parameter to the viWriteAsync() operation, no jobId will be returned. This option may be useful if only one asynchronous operation will be pending at a given time.

**OBSERVATION 6.1.16**

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

**PERMISSION 6.1.2**

The viWriteAsync() operation **MAY** be implemented synchronously, which could be done by using the viWrite() operation.

**RULE 6.1.17**

**IF** the viWriteAsync() operation is implemented synchronously, **AND** a given invocation of the operation is valid, **THEN** the operation **SHALL** return VI\_SUCCESS\_SYNC, **AND** all status information **SHALL** be returned in a VI\_EVENT\_IO\_COMPLETION.

**OBSERVATION 6.1.17**

The intent of PERMISSION 6.1.2 and RULE 6.1.14 is that an application can use the asynchronous operations transparently, even if the low-level driver used for a given VISA implementation supports only synchronous data transfers.

**RULE 6.1.18**

The status codes returned in the VI\_ATTR\_STATUS field of a VI\_EVENT\_IO\_COMPLETION event resulting from a call to viWriteAsync() **SHALL** be the same codes as those listed for viWrite().

**OBSERVATION 6.1.18**

The status code VI\_ERROR\_RSRC\_LOCKED can be returned either immediately or from the VI\_EVENT\_IO\_COMPLETION event.

**RULE 6.1.19**

For each successful call to viWriteAsync(), there **SHALL** be one and only one VI\_EVENT\_IO\_COMPLETION event occurrence.

**RULE 6.1.20**

**IF** the jobId parameter returned from viWriteAsync() is passed to viTerminate(), **AND** a VI\_EVENT\_IO\_COMPLETION event has not yet occurred for the specified jobId, **THEN** the viTerminate() operation **SHALL** raise a VI\_EVENT\_IO\_COMPLETION event on the given vi, **AND** the VI\_ATTR\_STATUS field of that event **SHALL** be set to VI\_ERROR\_ABORT.

**RULE 6.1.21**

**IF** the output parameter jobId is not VI\_NULL **AND** the return status from viWriteAsync() is successful, **THEN** the value in jobId **SHALL NOT** be VI\_NULL.

**OBSERVATION 6.1.19**

The value VI\_NULL is a reserved jobId and has a special meaning in viTerminate().

6.1.6 viWriteFromFile(vi, fileName, count, retCount)

**Purpose**

Take data from a file and write it out synchronously.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Transfer completed. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_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. |
| (continues) | |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_NCIC | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | No Listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |
| VI\_ERROR\_FILE\_ACCESS | An error occurred while trying to open the specified file. Possible reasons include an invalid path or lack of access rights. |
| VI\_ERROR\_FILE\_IO | An error occurred while accessing the specified file. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

This write operation synchronously transfers data. The file specified in fileName is opened in binary read-only mode, and the data (up to end-of-file or the number of bytes specified in count) is read. The data is then written to the device. This operation returns only when the transfer terminates.

This operation is useful for sending data that was already processed and/or formatted.

Table 6.1.6 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viWrite() and viReadToFile().

**Implementation Requirements**

**RULE 6.1.22**

The operation viWriteFromFile() **SHALL** open the file specified by fileName in binary mode.

**OBSERVATION 6.1.20**

If a VISA implementation uses the ANSI C file operations, the mode used by viWriteFromFile() should be “rb”.

**OBSERVATION 6.1.21**

If you pass VI\_NULL as the retCount parameter to the viWriteFromFile() operation, the number of bytes transferred will not be returned. This may be useful if it is important to know only whether the operation succeeded or failed.

6.1.7 viAssertTrigger(vi, protocol)

**Purpose**

Assert software or hardware trigger.

**Parameter**s

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to 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, VI\_TRIG\_PROT\_SYNC, VI\_TRIG\_PROT\_RESERVE, and VI\_TRIG\_PROT\_UNRESERVE. |

**Return Value**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | The specified trigger was successfully asserted to the device. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this 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\_PROT | The protocol specified is invalid. |
| 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 during transfer. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_LINE\_IN\_USE | The specified trigger line is currently in use. |

(continued)

|  |  |  |
| --- | --- | --- |
| **Error Codes** | | **Description** |
| VI\_ERROR\_NCIC | | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | | No Listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_INV\_SETUP | | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_CONN\_LOST | | The I/O connection for the given session has been lost. |
| VI\_ERROR\_LINE\_NRESERVED | An attempt was made to use a line that was not reserved. |

**Description**

This operation will source a software or hardware trigger dependent on the interface type. For a GPIB device, the device is addressed to listen, and then the GPIB *GET* command is sent. For a VXI device, if VI\_ATTR\_TRIG\_ID is VI\_TRIG\_SW, then the device is sent the Word Serial *Trigger* command; for any other values of the attribute, a hardware trigger is sent on the line corresponding to the value of that attribute. For a session to a Serial device or TCP/IP socket, if VI\_ATTR\_IO\_PROT is VI\_PROT\_4882\_STRS, the device is sent the string “\*TRG\n”; otherwise, this operation is not valid. For a session to a USB instrument, this function sends the TRIGGER message ID on the Bulk-OUT pipe.

For GPIB, ASRL, USB, and VXI software triggers, VI\_TRIG\_PROT\_DEFAULT is the only valid protocol. For VXI hardware triggers, VI\_TRIG\_PROT\_DEFAULT is equivalent to VI\_TRIG\_PROT\_SYNC.

For a PXI resource, viAssertTrigger() will reserve a trigger line for assertion, or release such a reservation. Instrument drivers should use viAssertTrigger() to ensure that they have ownership of a trigger line before performing any operation that could drive a signal onto that trigger line. The protocol parameter can be either VI\_TRIG\_PROT\_RESERVE or VI\_TRIG\_PROT\_UNRESERVE, which reserve a trigger line and release the reservation, respectively.

**Related Items**

See the INSTR resource description.

**Implementation Requirements**

**RULE 6.1.23**

For compatibility with earlier versions of this specification, VI\_TRIG\_PROT\_DEFAULT **SHALL** be equal to VI\_NULL.

**RULE 6.1.24**

**IF** the attribute VI\_ATTR\_IO\_PROT is set to VI\_PROT\_NORMAL for a session to an ASRL INSTR or TCPIP SOCKET resource, **THEN** the operation viAssertTrigger() **SHALL** return VI\_ERROR\_INV\_SETUP.

**RULE 6.1.25**

An INSTR resource implementation of viAssertTrigger() for a USB System **SHALL** return the error VI\_ERROR\_INV\_SETUP for a USBTMC base-class (non-488) device.

**RULE 6.1.26**

An INSTR resource implementation of viAssertTrigger() for a USB System **SHALL** return the error VI\_ERROR\_INV\_SETUP for a USBTMC 488-class device that does not implement the optional trigger message ID.

6.1.8 viReadSTB(vi, status)

**Purpose**

Read a status byte of the service request.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |  |
| --- | --- | --- |
| **Error Codes** | | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_SRQ\_NOCCURRED | | Service request has not been received for the session. |
| 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\_BERR | | Bus error occurred during transfer. |
| VI\_ERROR\_NCIC | | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | | No Listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). | |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. | |

**Description**

This operation reads a service request status from a service requester (the message-based device). For example, on the IEEE 488.2 interface, the message is read by polling devices; for other types of interfaces, a message is sent in response to a service request to retrieve status information. For a session to a Serial device or TCP/IP socket, if VI\_ATTR\_IO\_PROT is VI\_PROT\_4882\_STRS, the device is sent the string “\*STB?\n”, and then the device’s status byte is read; otherwise, this operation is not valid. If the status information is only one byte long, the most significant byte is returned with the zero value. If the service requester does not respond in the actual timeout period, VI\_ERROR\_TMO is returned. For a session to a USB instrument, this function sends the READ\_STATUS\_BYTE command on the control pipe.

**Related Items**

See the INSTR resource description.

**Implementation Requirements**

**RULE 6.1.27**

**IF** the attribute VI\_ATTR\_IO\_PROT is set to VI\_PROT\_NORMAL for a session to an ASRL INSTR or TCPIP SOCKET resource, **THEN** the operation viReadSTB() **SHALL** return VI\_ERROR\_INV\_SETUP.

**RULE 6.1.28**

An INSTR resource implementation of viReadSTB() for a USB System **SHALL** return the error VI\_ERROR\_INV\_SETUP for a USBTMC base-class (non-488) device.

**RULE 6.1.29**

**IF** the interface associated with the USB INSTR session has previously sent a service request notification, **THEN** viReadSTB() **SHALL** use the status byte from that notification rather than sending a new READ\_STATUS\_BYTE request on the control pipe.

**PERMISSION 6.1.3**

Since the operation viReadSTB() for USB INSTR must retain knowledge of service request notifications, a vendor **MAY** implement either a queue of status bytes from previous notifications or a single cached status byte, where each received status byte is bit-ORed into the single cached status byte.

6.1.9 viClear(vi)

**Purpose**

Clear a device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_RAW\_WR\_PROT\_VIOL | Violation of raw write protocol occurred during transfer. |
| VI\_ERROR\_RAW\_RD\_PROT\_VIOL | Violation of raw read protocol occurred during transfer. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_NCIC | The interface associated with the given vi is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | No Listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

This operation performs an IEEE 488.1-style clear of the device. For VXI INSTR sessions, VISA must use the Word Serial Clear command. For GPIB INSTR sessions, VISA must use the Selected Device Clear command. For Serial INSTR sessions, VISA must flush (discard) the I/O output buffer, send a break, and then flush (discard) the I/O input buffer. For TCP/IP SOCKET sessions, VISA must flush (discard) the I/O buffers. For USB INSTR sessions, VISA must send the INITIATE\_CLEAR and CHECK\_CLEAR\_STATUS commands on the control pipe.

**Related Items**

See the INSTR resource description.

**Implementation Requirements**

**OBSERVATION 6.1.22**

An invocation of the viClear() operations on an INSTR Resource will discard the read and write buffers used by the formatted I/O services for that session.

**PERMISSION 6.1.4**

An implementation of the viClear() operation for a Serial INSTR resource or a TCP/IP SOCKET resource **MAY** also send the string “\*CLS\n” to the device. This is allowed for backward compatibility with earlier VISA specifications that required this behavior.

**OBSERVATION 6.1.23**

The viClear() operation will no longer return an error for a Serial INSTR resource or a TCP/IP SOCKET resource when the attribute VI\_ATTR\_IO\_PROT is set to VI\_PROT\_NORMAL.

6.2 Formatted I/O Services

6.2.1 viSetBuf(vi, mask, size)

**Purpose**

Set the size for the formatted I/O and/or serial communication buffer(s).

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Buffer size set successfully. |
| VI\_WARN\_NSUP\_BUF | The specified buffer is not supported. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_ALLOC | The system could not allocate the buffer(s) of the specified size because of insufficient system resources. |
| VI\_ERROR\_INV\_MASK | The system cannot set the buffer for the given mask. |

**Description**

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

|  |  |
| --- | --- |
| **Flag** | **Interpretation** |
| VI\_READ\_BUF | Formatted I/O read buffer. |
| VI\_WRITE\_BUF | Formatted I/O write buffer. |
| VI\_IO\_IN\_BUF | I/O communication receive buffer. |
| VI\_IO\_OUT\_BUF | I/O communication transmit buffer. |

For backward compatibility, VI\_IO\_IN\_BUF is the same as VI\_ASRL\_IN\_BUF, and VI\_IO\_OUT\_BUF is the same as VI\_ASRL\_OUT\_BUF.

**Related Items**

See the INSTR resource description. Also see viFlush().

**Implementation Requirements**

**RULE 6.2.1**

A call to viSetBuf() **SHALL** flush the session’s related buffer(s) (for input buffers discard until END; for output buffers flush to device).

**RULE 6.2.2**

The system-allocated buffer(s) for a given session **SHALL** be freed by the system on session termination.

**OBSERVATION 6.2.1**

The size of the buffer(s) can have effects on the transfer performance for formatted I/O and/or low-level communication.

**RULE 6.2.3**

**IF** an ASRL INSTR or TCPIP INSTR or TCPIP SOCKET resource does not support setting the size of the I/O receive buffer, **THEN** a call to viSetBuf() with the VI\_IO\_IN\_BUF mask **SHALL** return VI\_WARN\_NSUP\_BUF.

**RULE 6.2.4**

**IF** an ASRL INSTR or TCPIP INSTR or TCPIP SOCKET resource does not support setting the size of the I/O transmit buffer, **THEN** a call to viSetBuf() with the VI\_IO\_OUT\_BUF mask **SHALL** return VI\_WARN\_NSUP\_BUF.

**OBSERVATION 6.2.2**

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

6.2.2 viFlush(vi, mask)

**Purpose**

Manually flush the specified buffers associated with formatted I/O operations and/or serial communication.

**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. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Buffers flushed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform read/write 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. |

**Description**

The value of mask can be one of the following flags:

|  |  |
| --- | --- |
| **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 performany 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\_IO\_IN\_BUF | Discards the receive buffer contents (same as VI\_IO\_IN\_BUF\_DISCARD). |
| VI\_IO\_IN\_BUF\_DISCARD | Discard the receive buffer contents (does not performany I/O to the device). |
| VI\_IO\_OUT\_BUF | Flush the transmit buffer by writing all buffered data to the device. |
| VI\_IO\_OUT\_BUF\_DISCARD | Discard the transmit buffer contents (does not perform any I/O to the device). |

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

Notice that when using formatted I/O operations with a serial device, a flush of the formatted I/O buffers also causes the corresponding serial communication buffers to be flushed. For example, calling viFlush() with VI\_WRITE\_BUF also flushes the VI\_IO\_OUT\_BUF.

For backward compatibility, VI\_IO\_IN\_BUF is the same as VI\_ASRL\_IN\_BUF, VI\_IO\_IN\_BUF\_DISCARD is the same as VI\_ASRL\_IN\_BUF\_DISCARD, VI\_IO\_OUT\_BUF is the same as VI\_ASRL\_OUT\_BUF, and VI\_IO\_OUT\_BUF\_DISCARD is the same as VI\_ASRL\_OUT\_BUF\_DISCARD.

**Related Items**

See the INSTR resource description. Also see viSetBuf().

**Implementation Requirements**

**RULE 6.2.5**

**if** viFlush() is invoked on an empty buffer, **then** the VISA system **SHALL NOT** perform any actions on the buffer.

6.2.3 viPrintf(vi, writeFmt, arg1, arg2,...)

**Purpose**

Convert, format, and send the parameters arg1, arg2, ... to the device as specified by the format string.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Parameters were successfully formatted. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform write 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 system resources. |

**Description**

This operation sends data to a device as specified by the format string. Before sending the data, the operation formats the arg characters in the parameter list as specified in the writeFmt string. The viWrite() operation performs the actual low-level I/O to the device. As a result, you should not use the viWrite() and viPrintf() operations 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 format code.

**Special Formatting Characters**

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

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

**Format Specifiers**

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

%[modifiers]*format code*

where *format code* specifies the data type in which the argument is represented. Modifiers are optional codes that describe the target data.

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

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

**ANSI C Standard Modifiers**

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Supported with Format Code** | **Description** |
| An integer specifying *field width*. | d, f, s format codes | This specifies the minimum field width of the converted argument. If an argument is shorter than the *field width*, it will be padded on the left (or on the right if the - flag is present).  Special case:  For the @H, @Q, and @B flags, the *field width*  includes the #H, #!, and #B strings, respectively.  A \* 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*. |

(continues)

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Supported with Format Code** | **Description** |
| An integer specifying *precision*. | d, f, s format codes | The *precision* string consists of a string of decimal digits. A . (decimal point) must prefix the *precision* string. 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 format codes.  b. The exact number of digits after the decimal point in case of f format codes.  c. The maximum numbers of characters for the string (s) specifier.  d. Maximum number of significant digits for g format code.  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, ll, L, z, and Z are legal values. (z and Z are not ANSI C standard flags.) | h (d, b, B format codes)  l (d, f, b, B format codes)  L (f format code)  z, Z (b, B format codes) | 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 format code type.  b. The l modifier promotes the argument to a long or unsigned long.  c. The ll modifier promotes the argument to a long long or unsigned long long.  d. The L modifier promotes the argument to a long double parameter.  e. The z modifier promotes the argument to an array of floats.  f. The Z modifier promotes the argument to an array of doubles. |

**Enhanced Modifiers to ANSI C Standards**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | | **Supported with Format Code** | **Description** | |
| A comma (‘,’) followed by an integer *n*, where *n* represents the array size. | | %d (plus variants) and %f only | The corresponding argument is interpreted as a reference to the first element of an array of size *n*. The first *n* elements of this list are printed in the format specified by the format code.  An asterisk (‘\*’) may be present after the ‘,’ modifier, in which case an extra arg is used. This arg must be an integer representing the array size of the given type. | |
| @1 | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (for example, 123). |
| @2 | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (for example, 123.45). |
| @3 | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined NR3 compatible number. An NR3 number is a floating point number represented in an exponential form (for example, 1.2345E-67). |
| @H | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <HEXADECIMAL NUMERIC RESPONSE DATA>. The number is represented in a base of 16 form. Only capital letters should represent numbers. The number is of form "#H*XXX*..," where *XXX*.. is a hexadecimal number (for example, #HAF35B). |
| @Q | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <OCTAL NUMERIC RESPONSE DATA>. The number is represented in a base of eight form. The number is of the form "#Q*YYY*..," where *YYY*.. is an octal number (for example, #Q71234). |
| @B | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <BINARY NUMERIC RESPONSE DATA>. The number is represented in a base two form. The number is of the form "#B*ZZZ*..," where *ZZZ*.. is a binary number (for example, #B011101001). |

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

**Standard ANSI C Format Codes**

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

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

**d** Argument type: An integer.

|  |  |
| --- | --- |
| **Modifier** | **Interpretation** |
| Default functionality | Print 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 ll | arg is a long 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.

**Enhanced Format Codes**

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

|  |  |
| --- | --- |
| **Flag or Modifier** | **Interpretation** |
| Default functionality | The data block is sent as an IEEE 488.2 <DEFINITE LENGTH ARBITRARY BLOCK RESPONSE DATA>. A count (long integer) mustappear as a flag that specifies the number of elements (by default, bytes) in the block. A *field width* or *precision* modifier is not allowed with this format code. |
| \* (asterisk) | An asterisk may be present instead of the count. In such a case, two args are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second arg is a reference to the data block. The size of an element is determined by the optional length modifier (see below), default being byte width. |
| Length modifier h | The data block is assumed to be an array of unsigned short integers (16 bits). The countcorresponds to the number of words rather than bytes. The data is swapped and padded into standard IEEE 488.2 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 format, if native computer representation is different. |
| Length modifier ll | The data block is assumed to be an array of unsigned long long integers. The count corresponds to the number of longlongwords (64 bits). Each longlongword data is swapped and padded into standard IEEE 488.2 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 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 will be represented in IEEE 754 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 ENDindicator set after the last byte of the block.

**y** Argument type: A location of a block of binary data.

|  |  |
| --- | --- |
| **Flags or Modifiers** | **Interpretation** |
| Default functionality | The data block is sent as raw binary data. A count (long integer) mustappear as a flag that specifies the number of elements (by default, bytes) in the block. A *field width* or *precision* modifier is not allowed with this format code. |
| \* (asterisk) | An asterisk may be present instead of the count. In such a case, two args are used, the first of which is a long integer specifying the count of the number of elements in the data block. The second arg is a reference to the data block. The size of an element is determined by the optional length modifier (see below), default being byte width. |
| Length modifier h | The data block is an array of unsigned short integers (16 bits). The countcorresponds to the number of words rather than bytes. If the optional “!ol” byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different. |
| Length modifier l | The data block is an array of unsigned long integers (32 bits). The countcorresponds to the number of longwords rather than bytes. If the optional “!ol” byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different. |
| Length modifier ll | The data block is an array of unsigned long long integers (64 bits). The countcorresponds to the number of longlongwords rather than bytes. If the optional “!ol” byte order modifier is present, the data is sent in little endian format; otherwise, the data is sent in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate if native computer representation is different. |
| Byte order modifier !ob | Data is sent in standard IEEE 488.2 (big endian) format. This is the default behavior if neither “!ob” nor “!ol” is present. |
| Byte order modifier !ol | Data is sent in little endian format. |

**OBSERVATION 6.2.3**

The END indicator is not appended when LF(\n) is part of a binary data block, as with %b or %B.

**BNF Format for viPrintf()**

The following is the BNF format for the viPrintf() writeFmtstring:

<print\_fmt> := {<slashed\_special> | <conversion> | <ascii\_char> }\*

<slashed\_special> := "\n" | "\r" | "\\" | "\t" | <oct\_esc> | "\"

<oct\_esc> := "\"<oct\_digit> [ <oct\_digit> [ <oct\_digit>]]

<ascii\_char> := ASCII characters (other than backslash (\), percent (%), and NULL).

<conversion> := <fmt\_cod\_d> | <fmt\_cod\_f> | <fmt\_cod\_c> | <fmt\_cod\_b> |

<fmt\_cod\_B> | <fmt\_cod\_s> | <fmt\_cod\_e> | <fmt\_cod\_y> | "%%"

<fmt\_cod\_d>:= "%" [<numeric\_mod> ] [<field width> ]

["." <precision> ] [","<array\_size>] ["l" | “ll” | "h"] "d"

<fmt\_cod\_f> := "%" [<numeric\_mod> ] [<field\_width> ]

["." <precision>] [","<array\_size>] [ "l" |"L"] "f"

<fmt\_cod\_e> := "%" [<numeric\_mod> ] [<field\_width> ]

["." <precision>] [","<array\_size>] [ "l" |"L"] "e"

<fmt\_cod\_b> := "%" <array\_size> [ "h" | "l" | “ll” | "z" |"Z"] "b"

<fmt\_cod\_B> := "%" <array\_size> [ "h" | "l" | “ll” | "z" | "Z"] "B"

<fmt\_cod\_c> := "%c"

<fmt\_cod\_s> := "%" [<just\_mod>] [<field\_width>] ["."<precision>] "s"

<fmt\_cod\_y> := "%" <array\_size> [ <swap\_mod> ] [ "h" | "l" | “ll” ] "y"

<swap\_mod> := "!ob" | "!ol"

<numeric\_mod> := "-" | "+" | " " | "@1" | "@2" | "@3" | "@H" | "@Q" | "@B"

<just\_mod> := "-"

<field\_width> := <positive\_integer> | "\*"

<precision> :=<positive\_integer> | "\*"

<array\_size> := <positive\_integer> | "\*"

**Related Items**

See the INSTR resource description. Also see viVPrintf().

**Implementation Requirements**

**RULE 6.2.6**

There **shall** be a one-to-one correspondence between % format conversion and arg parameters, except under the following circumstances:

1. If a \* is present for the *field width* modifier, then another arg parameter is used. This parameter is an integer.

2. If a \* is present for the *precision* modifier, then another arg parameter is used. This parameter is an integer.

3. If a \* is present for the *array\_size* in the %b, %B, or %y conversion, then another argparameter is used. This parameter is a long integer.

4. If a \* is present for the *array\_size* in the %d or %f conversion, then another argparameter is used. This parameter is an integer.

**OBSERVATION 6.2.4**

Up to four argparameters may be required to satisfy a % format conversion request. In the case where multiple argsare required, they appear in the following order:

*- field width* (\*with %d, %f, or %s) if used

*- precision* (\* with %d, %f, or %s) if used

- *array\_size* (\* with %b, %B, %y, %d, or %f) if used

- value to convert

**OBSERVATION 6.2.5**

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

**RULE 6.2.7**

For ANSI Ccompatibility the following conversion codes **SHALL** also be supported for output codes. These codes are ‘i,’ ‘o,’ ‘u,’ ‘n,’ ‘x,’ ‘X,’ ‘e,’ ‘E,’ ‘g,’ ‘G,’ and ‘p.’ For further explanation of these conversion codes, see the ANSI CStandard.

6.2.4 viVPrintf(vi, writeFmt, params)

**Purpose**

Convert, format, and send params to the device as specified by the format string.

**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. |
| 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Parameters were successfully formatted. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform write 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 system resources. |

**Description**

This operation is similar to viPrintf(),except that the ViVAList parameters list provides the parameters rather than separate argparameters.

**Related Items**

See the INSTR resource description. Also see viPrintf().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.2.5 viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

**Purpose**

Same as viPrintf(), except the data is written to a user-specified buffer rather than the device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Parameters were successfully formatted. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_INV\_FMT | A format specifier in the writeFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | A format specifier in the writeFmt string is not supported. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation is similar to viPrintf(),except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

**Related Items**

See the INSTR resource description. Also see viPrintf().

**Implementation Requirements**

**RULE 6.2.8**

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

6.2.6 viVSPrintf(vi, buf, writeFmt, params)

**Purpose**

Same as viVPrintf(), except that the data is written to a user-specified buffer rather than a device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Parameters were successfully formatted. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_INV\_FMT | A format specifier in the writeFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | A format specifier in the writeFmt string is not supported. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation is similar to viVPrintf(),except that the output is not written to the device; it is written to the user-specified buffer. This output buffer will be NULL terminated.

**Related Items**

See the INSTR resource description. Also see viSPrintf() and viVPrintf().

**Implementation Requirements**

**RULE 6.2.9**

**IF** the viVSPrintf() operations outputs an END indicator before all the arguments are satisfied, **THEN** the rest of the writeFmt string **SHALL** be ignored and the buffer string will still be terminated by a NULL.

6.2.7 viBufWrite(vi, buf, count, retCount)

**Purpose**

Similar to viWrite(), except the data is written to the formatted I/O write buffer rather than directly to the device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_INV\_SETUP | Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |

**Description**

This operation is similar to viWrite() and does not perform any kind of data formatting. It differs from viWrite() in that the data is written to the formatted I/O write buffer (the same buffer as used by viPrintf() and related operations) rather than directly to the device. This operation can intermix with the viPrintf() operation, but mixing it with the viWrite() operation is discouraged.

Table 6.2.1 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viWrite() and viBufRead().

**Implementation Requirements**

**RULE 6.2.10**

**IF** the viBufWrite() operation returns VI\_ERROR\_TMO, **THEN** the write buffer for the specified session **SHALL** be cleared.

**OBSERVATION 6.2.6**

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

6.2.8 viScanf(vi, readFmt, arg1, arg2,...)

**Purpose**

Read, convert, and format data using the format specifier. Store the formatted data in the arg1, arg2 parameters.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Data was successfully read and formatted into arg parameter(s). |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform read 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 system resources. |

**Description**

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

The format string consists of a %, followed by optional modifier flags, followed by one of the format codes in that sequence. It is of the form

**%[modifier]format code**

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

The viScanf() operation accepts input until an END indicator is read or all the format specifiers in the readFmt string are satisfied. 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 operation.

**OBSERVATION 6.2.7**

The viRead() operation is used for the actual low-level read from the device. Therefore, viRead() should not be used in the same session with formatted I/O operations. Also, if multiple sessions using formatted I/O resources are connected to the same device, the actual low-level reads must be synchronized between themselves.

**OBSERVATION 6.2.8**

Notice that when an END indicator is received, not all arguments in the format string may be consumed. However, the operation still returns a successful completion code.

**RULE 6.2.11**

The formatted I/O read operations **SHALL** honor the state of the VI\_ATTR\_TERMCHAR\_EN attribute.

**OBSERVATION 6.2.9**

Although formatted I/O operations generally read until an END indicator is received, RULE 6.2.11 allows the user to also specify a termination character that, if read, will cause the formatted I/O operations to stop reading from the device.

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

**ANSI C Standard Modifiers**

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

**Enhanced Modifiers to ANSI C Standards**

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Supported with Format Codes** | **Description** |
| A comma (‘,’) followed by an integer *n*, where *n* represents the array size. | %d (plus variants) and %f only | The corresponding argument is interpreted as a reference to the first element of an array of size *n*. The first *n* elements of this list are printed in the format specified by the format code.  A number sign (‘#’) may be present after the ‘,’ modifier, in which case an extra arg is used. This arg must be an integer representing the array size of the given type. |
| @1 | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined NR1 compatible number, which is an integer without any decimal point (for example, 123). |
| @2 | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined NR2 compatible number. The NR2 number has at least one digit after the decimal point (for example, 123.45). |

(continues)

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Supported with Format Codes** | **Description** |
| @H | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <HEXADECIMAL NUMERIC RESPONSE DATA>. The number is represented in a base of sixteen form. Only capital letters should represent numbers. The number is of form "#H*XXX*..," where *XXX*.. is a hexadecimal number (for example, #HAF35B). |
| @Q | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <OCTAL NUMERIC RESPONSE DATA>. The number is represented in a base of eight form. The number is of the form "#Q*YYY*..," where *YYY*.. is an octal number (for example, #Q71234). |
| @B | %d (plus variants) and %f only | Converts to an IEEE 488.2 defined <BINARY NUMERIC RESPONSE DATA>. The number is represented in a base two form. The number is of the form "#B*ZZZ*..," where *ZZZ*.. is a binary number (for example, #B011101001). |

**Format Codes**

**ANSI C Format Codes**

**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 NULLcharacter is added at the end of the data block. |

**Note:** White space in the device input stream is *not*ignored.

**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 may be in either IEEE 488.2 formats <DECIMAL NUMERIC PROGRAM DATA>, also known as NRf; flexible numeric representation (NR1, NR2, NR3...); 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 long integer. |
| Length modifier ll | arg is a reference to a long 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 may 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 double or long double, 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 is a pointer to an integer specifying the maximum array size. The second should be a reference to an array. 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 that were copied into the user array, not counting the trailing NULL character, are stored back in the integer pointed to by the first argument. |

**Enhanced Format Codes**

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

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

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

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

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

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

**y** Argument type: A reference to a data array.

|  |  |
| --- | --- |
| **Flags or Modifiers** | **Interpretation** |
| Default functionality | The data block is read as raw binary data. The format specifier sequence should have a flag describing the *array size*, which will give a maximum count of the number of bytes (or words or longwords, depending on length modifiers) to be read from the device. If the *array size* contains a # sign, two arguments are used. The first argument read is a pointer to a long integer specifying the maximum number of elements that the array can hold. The second 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 data block is assumed to be a reference to an array of unsigned short integers (16 bits). The countcorresponds to the number of words rather than bytes. If the optional “!ol” byte order modifier is present, the data being read is assumed to be in little endian format; otherwise, the data being read is assumed to be in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate to native computer format |
| Length modifier l | The data block is assumed to be a reference to an array of unsigned long integers (32 bits). The countcorresponds to the number of longwords rather than bytes. If the optional “!ol” byte order modifier is present, the data being read is assumed to be in little endian format; otherwise, the data being read is assumed to be in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate to native computer format |
| Length modifier ll | The data block is assumed to be a reference to an array of unsigned long long integers (64 bits). The countcorresponds to the number of longlongwords rather than bytes. If the optional “!ol” byte order modifier is present, the data being read is assumed to be in little endian format; otherwise, the data being read is assumed to be in standard IEEE 488.2 format. Data will be byte swapped and padded as appropriate to native computer format |
| Byte order modifier !ob | The data being read is assumed to be in standard IEEE 488.2 format. This is the default behavior if neither “!ob” nor “!ol” is present. |
| Byte order modifier !ol | The data being read is assumed to be in little endian format. |

**BNF format for** viScanf()readFmt **string**

The following is the BNF format for the viScanf() readFmtstring:

<scan\_fmt> := {<slashed\_special> | <conversion> | <ascii\_char> } \*

<slashed \_special> := "\n" | "\r" | "\t" | "\\" | <oct \_esc> | "\"

<oct\_esc> := "\"<oct\_digit> [ <oct\_digit> [ <oct\_digit> ] ]

<ascii\_char> := Any ASCII character except slash (\) or percent (%).

<conversion> :=<fmt\_cod\_c> | <fmt\_cod\_d> | <fmt\_cod\_e> | <fmt\_cod\_b> |

<fmt\_cod\_f> | <fmt\_cod\_s> | <fmt\_cod\_t> | <fmt\_cod\_T> |

<fmt\_cod\_y> | "%%"

<fmt\_cod\_b> := "%" ["\*"] [<array\_size > ] ["h" | "l" | “ll” | "z" | "Z" ] "b"

<fmt\_cod\_c> := "%" ["\*"] [<field\_width> ] "c"

<fmt\_cod\_d>:= "%" ["\*"] [","<array\_size>] ["l" | “ll” | "h"] "d"

<fmt\_cod\_e> := "%" ["\*"] [","<array\_size>] ["l" | "L"] "e"

<fmt\_cod\_f> := "%" ["\*"] [","<array\_size>] ["l" | "L"] "f"

<fmt\_cod\_s> := "%" ["\*"] [<field\_width> ] "s"

<fmt\_cod\_t> := "%" ["\*"] [<field\_width> ] "t"

<fmt\_cod\_T> := "%" ["\*"] [<field\_width> ] "T"

<fmt\_cod\_y> := "%" ["\*"] <array\_size> [ <swap\_mod> ] [ "h" | "l" | “ll” ] "y"

<swap\_mod> := "!ob" | "!ol"

<field\_width>:= <positive\_integer> | "#"

<array\_size>:= <positive\_integer> | "#"

**Related Items**

See the INSTR resource description. Also see viVScanf().

**Implementation Requirements**

**RULE 6.2.12**

There **SHALL** be a one-to-one correspondence between % format conversions and arg parameters in formatted I/O read operations except under the following circumstances:

• If a \* is present, no argparameters are used.

• If a # is present instead of *field width,* two arg parameters are used. The first arg is a reference to an integer (%c, %s, %t, %T). This arg defines the maximum size of the string being read. The second arg points to the buffer that will store the read data.

• If a # is present instead of *array\_size,* two arg parameters are used. The first arg is a reference to an integer (%d, %f) or a reference to a long integer (%b, %y). This arg defines the number of elements in the array. The second arg points to the array that will store the read data.

**RULE 6.2.13**

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

**OBSERVATION 6.2.10**

The size of the string defined in RULE 6.2.9 includes the trailing NULL character.

**RULE 6.2.14**

For ANSI Ccompatibility the following conversion codes **SHALL** also be supported for input codes. These codes are ‘i,’ ‘o,’ ‘u,’ ‘n,’ ‘x,’ ‘X,’ ‘e,’ ‘E,’ ‘g,’ ‘G,’ ‘p,’ ‘[...],’ and ‘[^...].’ For further explanation of these conversion codes, see the ANSI CStandard.

**RULE 6.2.15**

**If** viScanf()or a related formatted I/O read operationperforms a read that times out without returning any data, **then** the read buffer **shall** be cleared before that operationreturns.

**OBSERVATION 6.2.11**

When viScanf() or a related formatted I/O read operationtimes out, the next call to that operationwill encounter the empty buffer and force a read from the device. Note that this also applies to the Formatted I/O operations like viVScanf() and viBufRead() but not the Basic I/O operation viRead().

**RULE 6.2.16**

**IF** there is no remaining data to be parsed in the internal buffer, **AND** a new call to viScanf is issued, **THEN** VISA **SHALL** attempt to read more data from the instrument.

**OBSERVATION 6.2.12**

Note that if an instrument returns a single piece of data such as “123\n” with an END indicator, the behavior is different if a user makes one call to viScanf with two numeric arguments versus two calls to viScanf each with one numeric argument. In the first case, OBSERVATION 6.2.8 points out that the single call will return VI\_SUCCESS even though argument #2 is ignored. In the second case, RULE 6.2.16 points out that call #2 will not be ignored but will in fact read more data (or time out trying to do so).

**OBSERVATION 6.2.13**

When there is data in the internal buffer, whether that data can be parsed depends on the format modifier. For example, assume that only a newline character remains in the internal buffer. If a user calls viScanf with a numeric argument such as %d, then the newline is treated as whitespace and is ignored. Thus, VISA will read more data. However, if a user calls viScanf with %c, then the newline is character data that can be parsed that will satisfy the argument. Thus, VISA will not read more data at that time.

6.2.9 viVScanf(vi, readFmt, params)

**Purpose**

Read, convert, and format data using the format specifier. Store the formatted data in params.

**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. |
| 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Data was successfully read and formatted into arg parameter(s). |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform read 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 system resources. |

**Description**

This operation is similar to viScanf(), except that the ViVAList parameters list provides the parameters rather than separate argparameters.

**Related Items**

See the INSTR resource description. Also see viScanf().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.2.10 viSScanf(vi, buf, readFmt, arg1, arg2, ...)

**Purpose**

Same as viScanf(), except that the data is read from a user-specified buffer instead of a device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Data was successfully read and formatted into arg parameter(s). |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_INV\_FMT | A format specifier in the readFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | A format specifier in the readFmt string is not supported. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation is similar to viScanf(), except that the data is read from a user-specified buffer rather than a device.

**Related Items**

See the INSTR resource description. Also see viScanf().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.2.11 viVSScanf(vi, buf, readFmt, params)

**Purpose**

Same as viVScanf(), except that the data is read from a user-specified buffer instead of a device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Data was successfully read and formatted into arg parameter(s). |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_INV\_FMT | A format specifier in the readFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | A format specifier in the readFmt string is not supported. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation is similar to viVScanf(), except that the data is read from a user-specified buffer rather than a device.

**Related Items**

See the INSTR resource description. Also see viSScanf() and viVScanf().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.2.12 viBufRead(vi, buf, count, retCount)

**Purpose**

Similar to viRead(), except that the operation uses the formatted I/O read buffer for holding data read from the device.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | The operation completed successfully and the END indicator was received (for interfaces that have END indicators). |
| VI\_SUCCESS\_TERM\_CHAR | The specified termination character was read. |
| VI\_SUCCESS\_MAX\_CNT | The number of bytes read is equal to count. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_IO | An unknown I/O error occurred during transfer. |

**Description**

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

Table 6.2.2 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INSTR resource description. Also see viWrite().

**Implementation Requirements**

**RULE 6.2.17**

The operation viBufRead() **SHALL** return the success codes VI\_SUCCESS, VI\_SUCCESS\_MAX\_CNT,

and VI\_SUCCESS\_TERM\_CHAR under the same conditions as viRead().

6.2.13 viQueryf(vi, writeFmt, readFmt, arg1, arg2,...)

**Purpose**

Perform a formatted write and read through a single operation invocation.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| writeFmt | IN | ViString | ViString describing the format of write arguments. |
| readFmt | IN | ViString | ViString describing the format of 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Successfully completed the Query operation. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform read/write operation because of I/O error. |
| VI\_ERROR\_TMO | Timeout occurred before read/write operation completed. |
| VI\_ERROR\_INV\_FMT | A format specifier in the writeFmt or readFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | The format specifier is not supported for current argument type. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation 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 operation is a combination of the viPrintf() and viScanf() operations. 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 operation returns the same VISA status codes as viPrintf(), viScanf(), and viFlush().

**Related Items**

See the INSTR resource description. Also see ViVQueryf().

**Implementation Requirements**

**RULE 6.2.18**

When ViQueryf()executes, the read buffer **SHALL** be flushed before viPrintf() (write portion) executes. After this sequence, the write buffer **SHALL** be flushed before viScanf() executes. Depending on the state of the session, one or both of the flushes may be a no-operation.

6.2.14 viVQueryf(vi, writeFmt, readFmt, params)

**Purpose**

Perform a formatted write and read through a single operation invocation.

**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 to 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 operational return status. It returns either a completion code or an error code as follows. |

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Successfully completed the Query operation. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_IO | Could not perform read/write operation because of I/O error. |
| VI\_ERROR\_TMO | Timeout occurred before read/write operation completed. |
| VI\_ERROR\_INV\_FMT | A format specifier in the writeFmt or readFmt string is invalid. |
| VI\_ERROR\_NSUP\_FMT | The format specifier is not supported for current argument type. |
| VI\_ERROR\_ALLOC | The system could not allocate a formatted I/O buffer because of insufficient system resources. |

**Description**

This operation is similar to ViQueryf(), except that the ViVAList parameters list provides the parameters rather than the separate argparameter list.

**Related Items**

See the INSTR resource description. Also see ViQueryf().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.3 Memory I/O Services

6.3.1 viIn8(vi, space, offset, val8)

6.3.2 viIn16(vi, space, offset, val16)

6.3.3 viIn32(vi, space, offset, val32)

6.3.4 viIn64(vi, space, offset, val64)

**Purpose**

Read in an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified memory space and offset.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| space | IN | ViUInt16 | Specifies the address space. (See table.) |
| offset | IN | ViBusAddress | Offset (in bytes) of the address or register from which to read. |
| val8, val16, val32, or val64 | OUT | ViUInt8, ViUInt16 ViUInt32, or ViUInt64 | Data read from bus (8 bits for viIn8(),16 bits for viIn16(),32 bits for viIn32(), and 64 bits for ViIn64()). |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_INV\_SPACE | Invalid address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid offset specified. |
| VI\_ERROR\_NSUP\_OFFSET | Specified offset is not accessible from this hardware. |
| VI\_ERROR\_NSUP\_WIDTH | Specified width is not supported by this hardware. |
| VI\_ERROR\_NSUP\_ALIGN\_OFFSET | The specified offset is not properly aligned for the access width of the operation. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |

**Description**

This operation, by using the specified address space, reads in 8, 16, 32, or 64 bits of data from the specified offset. This operation does not require viMapAddress() or viMapAddressEx() to be called prior to its invocation.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Address the A16 address space of VXI/MXI bus. |
| VI\_A24\_SPACE | Address the A24 address space of VXI/MXI bus. |
| VI\_A32\_SPACE | Address the A32 address space of VXI/MXI bus. |
| VI\_A64\_SPACE | Address the A64 address space of VXI/MXI bus. |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viOut8(), viOut16(), viOut32(), and viOut64().

**Implementation Requirements**

**RULE 6.3.1**

The viInXX() operations **SHALL NOT** fail due to the configured state of the hardware used by the low-level memory access operations viMapAddressXX(), viPeekXX(), and viPokeXX().

**OBSERVATION 6.3.1**

The high-level operations viInXX() operate successfully independently from the low-level operations (viMapAddressXX(), viPeekXX(), and viPokeXX()). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

**RULE 6.3.2**

The viInXX() operations **SHALL** detect and return VI\_ERROR\_BERR on VXI transfers that are acknowledged by the VXI BERR\* (bus error) signal.

**RULE 6.3.3**

All VXI accesses performed by the viIn8() operation **SHALL** be D08 reads.

**RULE 6.3.4**

All VXI accesses performed by the viIn16() operation **SHALL** be D16 reads.

**RULE 6.3.5**

All VXI accesses performed by the viIn32() operation **SHALL** be D32 reads.

**RULE 6.3.6**

All VXI accesses performed by the viIn64() operation **SHALL** be D64 reads.

**RULE 6.3.7**

All VXI accesses performed by the viIn16(), viIn32(), and viIn64() operations **SHALL** be in the byte order specified by VI\_ATTR\_SRC\_BYTE\_ORDER.

**INSTR Specific**

The offset is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.2**

Notice that offset specified in the viInXX() operations for an INSTR resource is the offset address relative to the device’s allocated address base for the corresponding address space specified. For example, if space specifies VI\_A16\_SPACE, then offset specifies the offset from the logical address base address of the VXI device specified. If space specifies VI\_A24\_SPACE or VI\_A32\_SPACE or VI\_A64\_SPACE, then offset specifies the offset from the base address of the VXI device’s memory space allocated by the VXI Resource Manager within VXI A24, A32. or A64 space.

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

The offset parameter specifies an absolute address.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

6.3.5 viOut8(vi, space, offset, val8)

6.3.6 viOut16(vi, space, offset, val16)

6.3.7 viOut32(vi, space, offset, val32)

6.3.8 viOut64(vi, space, offset, val64)

**Purpose**

Write an 8-bit, 16-bit, 32-bit, or 64-bit value to the specified memory space and offset.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| space | IN | ViUInt16 | Specifies the address space. (See table.) |
| offset | IN | ViBusAddress | Offset (in bytes) of the address or register to which to write. |
| val8, val16, val32, or val64 | IN | ViUInt8, ViUInt16, ViUInt32, or ViUInt64 | Data to write to bus (8 bits for viOut8(), 16 bits for viOut16(), 32 bits for viOut32(), and 64 bits for ViOut64()). |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_INV\_SPACE | Invalid address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid offset specified. |
| VI\_ERROR\_NSUP\_OFFSET | Specified offset is not accessible from this hardware. |
| VI\_ERROR\_NSUP\_WIDTH | Specified width is not supported by this hardware. |
| VI\_ERROR\_NSUP\_ALIGN\_OFFSET | The specified offset is not properly aligned for the access width of the operation. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |

**Description**

This operation, by using the specified address space, writes 8, 16, 32, or 64 bits of data to the specified offset. This operation does not require viMapAddress() to be called prior to its invocation.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Address the A16 address space of VXI/MXI bus. |
| VI\_A24\_SPACE | Address the A24 address space of VXI/MXI bus. |
| VI\_A32\_SPACE | Address the A32 address space of VXI/MXI bus. |
| VI\_A64\_SPACE | Address the A64 address space of VXI/MXI bus. |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viIn8(), viIn16(), viIn32(), and viIn64().

**Implementation Requirements**

**RULE 6.3.8**

The viOutXX() operations **SHALL NOT** fail due to the configured state of the hardware used by the low-level memory access operations viMapAddressXX(),viPeekXX(), and viPokeXX().

**OBSERVATION 6.3.3**

The high-level operations viOutXX() operate successfully independently from the low-level operations (viMapAddressXX(), viPeekXX(), and viPokeXX()). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

**RULE 6.3.9**

The viOutXX() operations **SHALL** detect and return VI\_ERROR\_BERR on VXI transfers that are acknowledged by the VXI BERR\* (bus error) signal.

**RULE 6.3.10**

All VXI accesses performed by the viOut8() operation **SHALL** be D08 writes.

**RULE 6.3.11**

All VXI accesses performed by the viOut16() operation **SHALL** be D16 writes.

**RULE 6.3.12**

All VXI accesses performed by the viOut32() operation **SHALL** be D32 writes.

**RULE 6.3.13**

All VXI accesses performed by the viOut64() operation **SHALL** be D64 writes.

**RULE 6.3.14**

All VXI accesses performed by the viOut16() and viOut32() and viOut64() operations **SHALL** be in the byte order specified by VI\_ATTR\_DEST\_BYTE\_ORDER.

**INSTR Specific**

The offset is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.4**

Notice that offset specified in the viOutXX() operations for an INSTR resource is the offset address relative to the device’s allocated address base for the corresponding address space specified. For example, if space specifies VI\_A16\_SPACE, then offset specifies the offset from the logical address base address of the VXI device specified. If space specifies VI\_A24\_SPACE or VI\_A32\_SPACE or VI\_A64\_SPACE, then offset specifies the offset from the base address of the VXI device’s memory space allocated by the VXI Resource Manager within VXI A24 or A32 or A64 space.

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

The offset parameter specifies an absolute address.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

6.3.9 viMoveIn8(vi, space, offset, length, buf8)

6.3.10 viMoveIn16(vi, space, offset, length, buf16)

6.3.11 viMoveIn32(vi, space, offset, length, buf32)

6.3.12 viMoveIn64(vi, space, offset, length, buf64)

6.3.13 viMoveIn8Ex(vi, space, offset64, length, buf8)

6.3.14 viMoveIn16Ex(vi, space, offset64, length, buf16)

6.3.15 viMoveIn32Ex(vi, space, offset64, length, buf32)

6.3.16 viMoveIn64Ex(vi, space, offset64, length, buf64)

**Purpose**

Move a block of data from the specified address space and offset to local memory in increments of 8, 16, 32, or 64 bits.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| Vi | IN | ViSession | Unique logical identifier to a session. |
| Space | IN | ViUInt16 | Specifies the address space. (See table.) |
| offset or offset64 | IN | ViBusAddress or ViBusAddress64 | Offset (in bytes) of the starting address or register from which to read. |
| length | IN | ViBusSize | Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, 32, or 64 bits). |
| buf8, buf16, buf32, or buf64 | OUT | ViAUInt8, ViAUInt16, ViAUInt32, or ViAUInt64 | Data read from bus. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SPACE | Invalid address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid offset specified. |
| VI\_ERROR\_NSUP\_OFFSET | Specified offset is not accessible from this hardware. |
| VI\_ERROR\_NSUP\_WIDTH | Specified width is not supported by this hardware. |
| VI\_ERROR\_INV\_LENGTH | Invalid length specified. |
| VI\_ERROR\_NSUP\_ALIGN\_OFFSET | The specified offset is not properly aligned for the access width of the operation. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |

**Description**

This operation, by using the specified address space, reads in blocks of 8, 16, 32, or 64 bit data from the specified offset. This operation does not require viMapAddress() or viMapAddressEx() to be called prior to its invocation.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Address the A16 address space of VXI/MXI bus. |
| VI\_A24\_SPACE | Address the A24 address space of VXI/MXI bus. |
| VI\_A32\_SPACE | Address the A32 address space of VXI/MXI bus. |
| VI\_A64\_SPACE | Address the A64 address space of VXI/MXI bus. |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viMoveOut8(), viMoveOut16(), viMoveOut32(), and viMoveOut64().

**Implementation Requirements**

**RULE 6.3.15**

The viMoveInXX() operations **SHALL NOT** fail due to the configured state of the hardware used by the low-level memory access operations viMapAddressXX(), viPeekXX(), or viPokeXX().

**OBSERVATION 6.3.5**

The high-level operations viMoveInXX() operate successfully independently from the low-level operations (viMapAddressXX(), viPeekXX(), and viPokeXX()). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

**RULE 6.3.16**

The viMoveInXX(), operations **SHALL** detect and return VI\_ERROR\_BERR on VXI transfers that are acknowledged by the VXI BERR\* (bus error) signal.

**RULE 6.3.17**

All VXI accesses performed by the viMoveIn8() and viMoveIn8Ex() operations **SHALL** be D08 reads.

**RULE 6.3.18**

All VXI accesses performed by the viMoveIn16() and viMoveIn16Ex() operations **SHALL** be D16 reads.

**RULE 6.3.19**

All VXI accesses performed by the viMoveIn32() and viMoveIn32Ex() operations **SHALL** be D32 reads.

**RULE 6.3.20**

All VXI accesses performed by the viMoveIn64() and viMoveIn64() operations **SHALL** be D64 reads.

**RULE 6.3.21**

All VXI accesses performed by the viMoveIn16(), viMoveIn32(), and viMoveIn64() operations **SHALL** be in the byte order specified by VI\_ATTR\_SRC\_BYTE\_ORDER.

**RULE 6.3.22**

All VISA implementations of the viMoveInXX() operations **SHALL** ignore the attribute VI\_ATTR\_DEST\_INCREMENT **AND SHALL** increment the local buffer address for each element.

**OBSERVATION 6.3.6**

It is valid for the VISA driver to copy the data into the user buffer at any width it wishes. In other words, even if the width is a byte (8-bit), the VISA driver is allowed to perform 32-bit PCI burst accesses since it is just memory, in order to improve throughput. It is also valid for other utilities to dereference the user buffer more than once, since it is not considered volatile.

**INSTR Specific**

The offset is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.7**

Notice that offset specified in the viMoveInXX() operations for an INSTR resource is the offset address relative to the device’s allocated address base for the corresponding address space specified. For example, if space specifies VI\_A16\_SPACE, then offset specifies the offset from the logical address base address of the VXI device specified. If space specifies VI\_A24\_SPACE or VI\_A32\_SPACE or VI\_A64\_SPACE, then offset specifies the offset from the base address of the VXI device’s memory space allocated by the VXI Resource Manager within VXI A24, A32, or A64 space.

**OBSERVATION 6.3.8**

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

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

The offset parameter specifies an absolute address.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

**OBSERVATION 6.3.9**

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

6.3.17 viMoveOut8(vi, space, offset, length, buf8)

6.3.18 viMoveOut16(vi, space, offset, length, buf16)

6.3.19 viMoveOut32(vi, space, offset, length, buf32)

6.3.20 viMoveOut64(vi, space, offset, length, buf64)

6.3.21 viMoveOut8Ex(vi, space, offset64, length, buf8)

6.3.22 viMoveOut16Ex(vi, space, offset64, length, buf16)

6.3.23 viMoveOut32Ex(vi, space, offset64, length, buf32)

6.3.24 viMoveOut64Ex(vi, space, offset64, length, buf64)

**Purpose**

Move a block of data from local memory to the specified address space and offset in increments of 8, 16, 32, or 64 bits.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| Vi | IN | ViSession | Unique logical identifier to a session. |
| space | IN | ViUInt16 | Specifies the address space. (See table.) |
| offset or offset64 | IN | ViBusAddress or ViBusAddress64 | Offset (in bytes) of the starting address or register to which to write. |
| length | IN | ViBusSize | Number of elements to transfer, where the data width of the elements to transfer is identical to data width (8, 16, 32, or 64 bits). |
| buf8, buf16, buf32, or buf64 | IN | ViAUInt8, ViAUInt16, ViAUInt32, or ViAUInt64 | Data to write to bus. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SPACE | Invalid address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid offset specified. |
| VI\_ERROR\_NSUP\_OFFSET | Specified offset is not accessible from this hardware. |
| VI\_ERROR\_NSUP\_WIDTH | Specified width is not supported by this hardware. |
| VI\_ERROR\_INV\_LENGTH | Invalid length specified. |
| VI\_ERROR\_NSUP\_ALIGN\_OFFSET | The specified offset is not properly aligned for the access width of the operation. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |

**Description**

This operation, by using the specified address space, writes blocks of 8, 16, 32, or 64 bit data to the specified offset. This operation does not require viMapAddress() or viMapAddressEx() to be called prior to its invocation.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Address the A16 address space of VXI/MXI bus. |
| VI\_A24\_SPACE | Address the A24 address space of VXI/MXI bus. |
| VI\_A32\_SPACE | Address the A32 address space of VXI/MXI bus. |
| VI\_A64\_SPACE | Address the A64 address space of VXI/MXI bus. |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viMoveIn8(), viMoveIn16(), viMoveIn32(), and viMoveIn64().

**Implementation Requirements**

**RULE 6.3.23**

The viMoveOutXX() operations **SHALL NOT** fail due to the configured state of the hardware used by the low-level memory access operations viMapAddressXX(), viPeekXX(), and viPokeXX().

**OBSERVATION 6.3.10**

The high-level operations viMoveOutXX() operate successfully independently from the low-level operations (viMapAddressXX(), viPeekXX(), and viPokeXX()). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

**RULE 6.3.24**

The viMoveOutXX() operations **SHALL** detect and return VI\_ERROR\_BERR on VXI transfers that are acknowledged by the VXI BERR\* (bus error) signal.

**RULE 6.3.25**

All VXI accesses performed by the viMoveOut8() and viMoveOut8Ex() operations **SHALL** be D08 writes.

**RULE 6.3.26**

All VXI accesses performed by the viMoveOut16() and viMoveOut16Ex() operations **SHALL** be D16 writes.

**RULE 6.3.27**

All VXI accesses performed by the viMoveOut32() and viMoveOut32Ex() operations **SHALL** be D32 writes.

**RULE 6.3.28**

All VXI accesses performed by the viMoveOut64() and viMoveOut64Ex() operations **SHALL** be D64 writes.

**RULE 6.3.29**

All VXI accesses performed by the viMoveOut16() and viMoveOut32() and viMoveOut64() operations **SHALL** be in the byte order specified by VI\_ATTR\_DEST\_BYTE\_ORDER.

**RULE 6.3.30**

All VISA implementations of the viMoveOutXX() operations **SHALL** ignore the attribute VI\_ATTR\_SRC\_INCREMENT **AND SHALL** increment the local buffer address for each element.

**OBSERVATION 6.3.11**

It is valid for the VISA driver to copy the data out of the user buffer at any width it wishes. In other words, even if the width is a byte (8-bit), the VISA driver is allowed to perform 32-bit PCI burst accesses since it is just memory, in order to improve throughput. It is also valid for other utilities to dereference the user buffer more than once, since it is not considered volatile.

**INSTR Specific**

The offset is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.12**

Notice that offset specified in the viMoveOutXX() operations for an INSTR resource is the offset address relative to the device’s allocated address base for the corresponding address space specified. For example, if space specifies VI\_A16\_SPACE, then offset specifies the offset from the logical address base address of the VXI device specified. If space specifies VI\_A24\_SPACE or VI\_A32\_SPACE or VI\_A64\_SPACE, then offset specifies the offset from the base address of the VXI device’s memory space allocated by the VXI Resource Manager within VXI A24, A32, or A64 space.

**OBSERVATION 6.3.13**

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

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

The offset parameter specifies an absolute address.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

**OBSERVATION 6.3.14**

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

6.3.25 viMove(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length)

6.3.26 viMoveEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length)

**Purpose**

Move a block of data.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| srcSpace | IN | ViUInt16 | Specifies the address space of the source. |
| srcOffset or srcOffset64 | IN | ViBusAddress or ViBusAddres64 | Offset of the starting address or register from which to read. |
| srcWidth | IN | ViUInt16 | Specifies the data width of the source. |
| destSpace | IN | ViUInt16 | Specifies the address space of the destination. |
| destOffset or destOffset64 | IN | ViBusAddress or ViBusAddress64 | Offset of the starting address or register to which to write. |
| destWidth | IN | ViUInt16 | Specifies the data width of the destination. |
| length | IN | ViBusSize | Number of elements to transfer, where the data width of the elements to transfer is identical to source data width. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_INV\_SPACE | Invalid source or destination address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid source or destination offset specified. |
| VI\_ERROR\_INV\_WIDTH | Invalid source or destination width specified. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_NSUP\_OFFSET | Specified source or destination offset is not accessible from this hardware. |
| VI\_ERROR\_NSUP\_VAR\_WIDTH | Cannot support source and destination widths that are different. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_NSUP\_WIDTH | Specified width is not supported by this hardware. |
| VI\_ERROR\_NSUP\_ALIGN\_OFFSET | The specified offset is not properly aligned for the access width of the operation. |
| VI\_ERROR\_INV\_LENGTH | Invalid length specified. |

**Description**

This operation moves data from the specified source to the specified destination. The source and the destination can either be local memory or the offset of the interface with which this MEMACC Resource is associated. This operation uses the specified data width and address space. In some systems, such as VXI, users can specify additional settings for the transfer, like byte order and access privilege, by manipulating the appropriate attributes.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Addresses the A16 address space of the VXI/MXI bus. |
| VI\_A24\_SPACE | Addresses the A24 address space of the VXI/MXI bus. |
| VI\_A32\_SPACE | Addresses the A32 address space of the VXI/MXI bus. |
| VI\_A64\_SPACE | Addresses the A64 address space of the VXI/MXI bus. |
| VI\_LOCAL\_SPACE | Addresses process-local memory (using a virtual address). |
| VI\_OPAQUE\_SPACE | Addresses potentially volatile data (using a virtual address). |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

The following table lists the valid entries for specifying widths.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_WIDTH\_8 | Performs 8-bit (D08) transfers. |
| VI\_WIDTH\_16 | Performs 16-bit (D16) transfers. |
| VI\_WIDTH\_32 | Performs 32-bit (D32) transfers. |
| VI\_WIDTH\_64 | Performs 64-bit (D64) transfers. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viMoveAsync() and viMoveAsyncEx().

**Implementation Requirements**

**RULE 6.3.31**

The viMove() and viMoveEx() operations **SHALL NOT** fail due to the configured state of the hardware used by the low-level memory access operations viMapAddressXX(), viPeekXX(), and viPokeXX().

**OBSERVATION 6.3.15**

The high-level operations viMove() and viMoveEx() operate successfully independently from the low-level operations (viMapAddressXX(), viPeekXX(), and viPokeXX()). The high-level and low-level operations should operate independently regardless of the configured state of the hardware that is used to perform memory accesses.

**RULE 6.3.32**

The viMove() and viMoveEx() operations **SHALL** detect and return VI\_ERROR\_BERR on VXI transfers that are acknowledged by the VXI BERR\* (bus error) signal.

**RULE 6.3.33**

All VXI accesses performed by the viMove() and viMoveEx() operations **SHALL** be in the byte order specified by VI\_ATTR\_SRC\_BYTE\_ORDER and VI\_ATTR\_DEST\_BYTE\_ORDER.

**OBSERVATION 6.3.16**

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

**RULE 6.3.34**

**IF** srcSpace is VI\_LOCAL\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** ignore VI\_ATTR\_SRC\_BYTE\_ORDER.

**RULE 6.3.35**

**IF** destSpace is VI\_LOCAL\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** ignore VI\_ATTR\_DEST\_BYTE\_ORDER.

**OBSERVATION 6.3.17**

Local accesses use the native byte order rather than the byte order specified by the attributes.

**RULE 6.3.36**

All VXI accesses performed by the viMove() and viMoveEx() operations **SHALL** use either the same or successive offsets, depending on the increment value specified by VI\_ATTR\_SRC\_INCREMENT and VI\_ATTR\_DEST\_INCREMENT.

**RULE 6.3.37**

**IF** srcSpace is VI\_LOCAL\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** ignore VI\_ATTR\_SRC\_INCREMENT.

**RULE 6.3.38**

**IF** destSpace is VI\_LOCAL\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** ignore VI\_ATTR\_DEST\_INCREMENT.

**OBSERVATION 6.3.18**

Local accesses always increment the offset for each index in a multi-element transfer, rather than using the increment specified by the attributes.

**RULE 6.3.39**

**IF** srcSpace is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** honor VI\_ATTR\_SRC\_INCREMENT.

**RULE 6.3.40**

**IF** destSpace is any value other than VI\_LOCAL\_SPACE, including VI\_OPAQUE\_SPACE, **THEN** viMove() and viMoveEx() **SHALL** honor VI\_ATTR\_DEST\_INCREMENT.

**OBSERVATION 6.3.19**

While VI\_OPAQUE\_SPACE uses a process-local virtual address, it is not necessarily pointing to system memory, so it may be a FIFO. Therefore, VI\_ATTR\_SRC/DEST\_INCREMENT do indeed apply. The VISA driver must copy the data using the specified width. Other utilities may not dereference the pointer since it should be considered volatile.

**INSTR Specific**

If srcSpace is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, then srcOffset is a relative address of the device associated with the given INSTR resource. Similarly, if destspace is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, then destOffset is a relative address of the device associated with the given INSTR resource.

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

**OBSERVATION 6.3.20**

Notice that srcOffset, destOffset, srcOffset64, and destOffset64 specified in the viMove() and viMoveEx() operations for a MEMACC resource are absolute addresses.

6.3.27 viMoveAsync(vi, srcSpace, srcOffset, srcWidth, destSpace, destOffset, destWidth, length, jobId)

6.3.28 viMoveAsyncEx(vi, srcSpace, srcOffset64, srcWidth, destSpace, destOffset64, destWidth, length, jobId)

**Purpose**

Move a block of data asynchronously.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| srcSpace | IN | ViUInt16 | Specifies the address space of the source. |
| srcOffset or srcOffset64 | IN | ViBusAddress or ViBusAddress64 | Offset of the starting address or register from which to read. |
| srcWidth | IN | ViUInt16 | Specifies the data width of the source. |
| destSpace | IN | ViUInt16 | Specifies the address space of the destination. |
| destOffset or destOffset64 | IN | ViBusAddress or ViBusAddress64 | Offset of the starting address or register to which to write. |
| destWidth | IN | ViUInt16 | Specifies the data width of the destination. |
| length | IN | ViBusSize | Number of elements to transfer, where the data width of the elements to transfer is identical to source data width. |
| jobId | OUT | ViJobId | Represents the location of an integer that will be set to the job identifier of this asynchronous move operation. Each time an asynchronous move operation is called, it is assigned a unique job identifier. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Asynchronous operation successfully queued. |
| VI\_SUCCESS\_SYNC | Operation performed synchronously. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_QUEUE | Unable to queue move operation. |
| VI\_ERROR\_IN\_PROGRESS | Unable to start a new asynchronous operation while another asynchronous operation is in progress. |

**Description**

This operation asynchronously moves data from the specified source to the specified destination. This operation queues up the transfer in the system, then it returns immediately without waiting for the transfer to carry out or complete. When the transfer terminates, a VI\_EVENT\_IO\_COMPLETION event indicates the status of the transfer.

The operation returns jobId, which you can use either with viTerminate() to abort the operation or with VI\_EVENT\_IO\_COMPLETION events to identify which asynchronous move operations completed.

The source and the destination can be either local memory or the offset of the device/interface with which this INSTR or MEMACC Resource is associated. This operation uses the specified data width and address space. In some systems, such as VXI, users can specify additional settings for the transfer, like byte order and access privilege, by manipulating the appropriate attributes.

The following table lists the valid entries for specifying address space.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Addresses the A16 address space of the VXI/MXI bus. |
| VI\_A24\_SPACE | Addresses the A24 address space of the VXI/MXI bus. |
| VI\_A32\_SPACE | Addresses the A32 address space of the VXI/MXI bus. |
| VI\_LOCAL\_SPACE | Addresses process-local memory (using a virtual address). |
| VI\_OPAQUE\_SPACE | Addresses potentially volatile data (using a virtual address). |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

The following table lists the valid entries for specifying widths.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_WIDTH\_8 | Performs 8-bit (D08) transfers. |
| VI\_WIDTH\_16 | Performs 16-bit (D16) transfers. |
| VI\_WIDTH\_32 | Performs 32-bit (D32) transfers. |
| VI\_WIDTH\_64 | Performs 64-bit (D64) transfers. |

Table 6.3.1 Special Values for jobId Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return a job identifier. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viMove().

**Implementation Requirements**

**RULE 6.3.41**

**IF** the output parameter jobId is not VI\_NULL, **THEN** the value in jobId **SHALL** be valid before viMoveAsync() begins the data transfer.

**OBSERVATION** **6.3.21**

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

**OBSERVATION** **6.3.22**

If you pass VI\_NULL as the jobId parameter to the viMoveAsync() operation, no jobId will be returned. This option may be useful if only one asynchronous operation will be pending at a given time.

**OBSERVATION** **6.3.23**

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

**PERMISSION** **6.3.1**

The viMoveAsync() operation **MAY** be implemented synchronously, which could be done by using the viMove() operation.

**RULE 6.3.42**

**IF** the viMoveAsync() operation is implemented synchronously, **AND** a given invocation of the operation is valid, **THEN** the operation **SHALL** return VI\_SUCCESS\_SYNC, **AND** all status information **SHALL** be returned in a VI\_EVENT\_IO\_COMPLETION.

**OBSERVATION** **6.3.24**

The intent of PERMISSION 6.3.1 and RULE 6.3.42 is that an application can use the asynchronous operations transparently, even if the low-level driver used for a given VISA implementation supports only synchronous data transfers.

**RULE 6.3.43**

The status codes returned in the VI\_ATTR\_STATUS field of a VI\_EVENT\_IO\_COMPLETION event resulting from a call to viMoveAsync() **SHALL** be the same codes as those listed for viMove().

**OBSERVATION** **6.3.25**

The status code VI\_ERROR\_RSRC\_LOCKED can be returned either immediately or from the VI\_EVENT\_IO\_COMPLETION event.

**RULE 6.3.44**

For each successful call to viMoveAsync(), there **SHALL** be one and only one VI\_EVENT\_IO\_COMPLETION event occurrence.

**RULE 6.3.45**

**IF** the jobId parameter returned from viMoveAsync() is passed to viTerminate(), **AND** a VI\_EVENT\_IO\_COMPLETION event has not yet occurred for the specified jobId, **THEN** the viTerminate() operation **SHALL** raise a VI\_EVENT\_IO\_COMPLETION event on the given vi, **AND** the VI\_ATTR\_STATUS field of that event **SHALL** be set to VI\_ERROR\_ABORT.

**RULE 6.3.46**

**IF** the output parameter jobId is not VI\_NULL **AND** the return status from viMoveAsync() is successful, **THEN** the value in jobId **SHALL NOT** be VI\_NULL.

**OBSERVATION 6.3.26**

The value VI\_NULL is a reserved jobId and has a special meaning in viTerminate().

**INSTR Specific**

If srcSpace is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, then srcOffset is a relative address of the device associated with the given INSTR resource. Similarly, if destspace is neither VI\_LOCAL\_SPACE nor VI\_OPAQUE\_SPACE, then destOffset is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.27**

The primary intended use of this operation with an INSTR session is to asynchronously move data to or from the device. Therefore, either the srcSpace or destSpace parameter will usually be VI\_LOCAL\_SPACE.

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

6.3.29 viMapAddress(vi, mapSpace, mapBase, mapSize, access, suggested, address)

6.3.30 viMapAddressEx(vi, mapSpace, mapBase64, mapSize, access, suggested, address)

**Purpose**

Map the specified memory space into the process’s address space.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| mapSpace | IN | ViUInt16 | Specifies the address space to map. |
| mapBase or mapBase64 | IN | ViBusAddress or ViBusAddress64 | 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 operation may map the memory into an address region different from suggested. |
| address | OUT | ViAddr | Address in your process space where the memory was mapped. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Map successful. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this 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\_SPACE | Invalid address space specified. |
| VI\_ERROR\_INV\_OFFSET | Invalid offset specified. |

(continues)

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_NSUP\_OFFSET | Specified region is not accessible from this hardware. |
| VI\_ERROR\_INV\_SIZE | Invalid size of window specified. |
| VI\_ERROR\_INV\_ACC\_MODE | Invalid access mode. |
| VI\_ERROR\_TMO | viMapAddress() could not acquire resource or perform mapping before the timer expired. |
| VI\_ERROR\_ALLOC | Unable to allocate window of at least the requested size. |
| VI\_ERROR\_WINDOW\_MAPPED | The specified session already contains a mapped window. |
| VI\_ERROR\_INV\_SETUP | Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state). |

**Description**

This operation 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 (refer to the following table) parameter. The address parameter returns the address in your process space where memory is mapped.

The following table lists the valid entries for the mapSpace parameter.

|  |  |
| --- | --- |
| **Value** | **Description** |
| VI\_A16\_SPACE | Map the A16 address space of VXI/MXI bus. |
| VI\_A24\_SPACE | Map the A24 address space of VXI/MXI bus. |
| VI\_A32\_SPACE | Map the A32 address space of VXI/MXI bus. |
| VI\_A64\_SPACE | Map the A64 address space of VXI/MXI bus. |
| VI\_PXI\_CFG\_SPACE | Address the PCI configuration space. |
| VI\_PXI\_BAR0\_SPACE – VI\_PXI\_BAR5\_SPACE | Address the specified PCI memory or I/O space. |
| VI\_PXI\_ALLOC\_SPACE | Access physical locally allocated memory. |

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viUnmapAddress().

**Implementation Requirements**

**RULE 6.3.47**

**IF** a call to viMapAddress() or viMapAddressEx() succeeds, **THEN** the value of VI\_ATTR\_WIN\_ACCESS for the given vi **SHALL** be set to either VI\_USE\_OPERS or VI\_DEREF\_ADDR.

**RULE 6.3.48**

**IF** the value of VI\_ATTR\_RSRC\_SPEC\_VERSION is greater than or equal to 0x00100100, **AND** a call to viMapAddress() or viMapAddressEx() succeeds, **AND** the value of the address parameter   
cannot be directly dereferenced such that all VXI accesses are in the byte order specified by VI\_ATTR\_WIN\_BYTE\_ORDER, **THEN** the value of VI\_ATTR\_WIN\_ACCESS for the given vi **SHALL** be set to VI\_USE\_OPERS.

**INSTR Specific**

The mapBase or mapBase64 is a relative address of the device associated with the given INSTR resource.

**OBSERVATION 6.3.28**

Notice that mapBaseXX specified in the viMapAddressXX() operation for an INSTR resource is the offset address relative to the device’s allocated address base for the corresponding address space specified. For example, if mapSpace specifies VI\_A16\_SPACE, then mapBase specifies the offset from the logical address base address of the VXI device specified. If mapSpace specifies VI\_A24\_SPACE or VI\_A32\_SPACE or VI\_A64\_SPACE, then mapBase specifies the offset from the base address of the VXI device’s memory space allocated by the VXI Resource Manager within VXI A24 or A32 or A64 space.

All operations on a PXI INSTR resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following values for the space parameter: VI\_PXI\_CFG\_SPACE, VI\_PXI\_BAR0\_SPACE, VI\_PXI\_BAR1\_SPACE, VI\_PXI\_BAR2\_SPACE, VI\_PXI\_BAR3\_SPACE, VI\_PXI\_BAR4\_SPACE, and VI\_PXI\_BAR5\_SPACE.

**MEMACC Specific**

The mapBaseXX parameter specifies an absolute address.

All operations on a PXI MEMACC resource that accept a space parameter to indicate the address space for bus access **SHALL** accept the following value for the space parameter: VI\_PXI\_ALLOC\_SPACE.

6.3.31 viUnmapAddress(vi)

**Purpose**

Unmap memory space previously mapped by viMapAddress() or viMapAddressEx().

**Parameter**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_WINDOW\_NMAPPED | The specified session is not currently mapped. |

**Description**

This operation unmaps the region previously mapped by the viMapAddress() operation.

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viMapAddress().

**Implementation Requirements**

**RULE 6.3.49**

**IF** a call to viUnmapAddress() succeeds, **THEN** the value of VI\_ATTR\_WIN\_ACCESS for the given vi **SHALL** be set to VI\_NMAPPED.

6.3.32 viPeek8(vi, addr, val8)

6.3.33 viPeek16(vi, addr, val16)

6.3.34 viPeek32(vi, addr, val32)

6.3.35 viPeek64(vi, addr, val64)

**Purpose**

Read an 8-bit, 16-bit, 32-bit, or 64-bit value from the specified address.

**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, val32, or val64 | OUT | ViUInt8, ViUInt16, ViUInt32, or ViUInt64 | Data read from bus (8 bits for viPeek8(), 16 bits for viPeek16(),32 bits for viPeek32(), and 64 bits for viPeek64()). |

**Return Values**

None

**Description**

This operation reads an 8-bit, 16-bit, 32-bit, or 64-bit value from the address location specified in addr. The address must be a valid memory address in the current process mapped by a previous viMapAddress() or viMapAddressEx() call.

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viPoke8(), viPoke16(), viPoke32(), and viPoke64().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.3.36 viPoke8(vi, addr, val8)

6.3.37 viPoke16(vi, addr, val16)

6.3.38 viPoke32(vi, addr, val32)

6.3.39 viPoke64(vi, addr, val64)

**Purpose**

Write an 8-bit, 16-bit, 32-bit, or 64-bit value to the specified address.

**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, val32, or val64 | IN | ViUInt8, ViUInt16, ViUInt32, or ViUInt64 | Data to write to bus (8 bits for viPoke8(), 16 bits for viPoke16(), 32 bits for viPoke32(), and 64 bits for viPoke64()). |

**Return Values**

None

**Description**

This operation takes an 8-bit, 16-bit, 32-bit, or 64-bit value and stores its content to the address pointed to by addr. The address must be a valid memory address in the current process mapped by a previous viMapAddress() or viMapAddressEx() call.

**Related Items**

See the INSTR and MEMACC resource descriptions. Also see viPeek8(), viPeek16(), viPeek32(), and viPeek64().

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.4 Shared Memory Services

6.4.1 viMemAlloc(vi, size, offset)

6.4.2 viMemAllocEx(vi, size, offset64)

**Purpose**

Allocate memory from a device’s memory region.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| size | IN | ViBusSize | Specifies the size of the allocation. |
| offset or offset64 | OUT | ViBusAddress or ViBusAddress64 | Returns the offset of the allocated device memory. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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. |

**Description**

This operation 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.

**Related Items**

See the INSTR resource description. Also see viMemFree() and viMemFreeEx().

**Implementation Requirements**

**OBSERVATION 6.4.1**

Notice that offset returned from the viMemAlloc() and viMemAllocEx() operations is the offset address relative to the device’s allocated address base for whichever address space into which the given device exports memory.

**OBSERVATION 6.4.2**

No device is required to have memory that can be shared or managed by the local controller. In this case, a VISA implementation may always return VI\_ERROR\_NSUP\_OPER.

**RULE 6.4.1**

The offset parameter in the viMemAlloc(), viMemAllocEx(), viMemFree(), and viMemFreeEx() operations on a PXI MEMACC resource **SHALL** be an absolute physical PCI address.

6.4.3 viMemFree(vi, offset)

6.4.4 viMemFreeEx(vi, offset64)

**Purpose**

Free memory previously allocated using viMemAlloc() or viMemAllocEx().

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| Vi | IN | ViSession | Unique logical identifier to a session. |
| offset or offset64 | IN | ViBusAddress or ViBusAddress64 | Specifies the memory previously allocated with viMemAlloc() or viMemAllocEx(). |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_MAPPED | The specified offset is currently in use by viMapAddress(). |

**Description**

This operation frees the memory previously allocated using viMemAlloc() or viMemAllocEx().

**Related Items**

See the INSTR resource description. Also see viMemAlloc(), and viMemAllocEx().

**Implementation Requirements**

**RULE 6.4.2**

**IF** the offset parameter specifies a valid address that was previously allocated using the viMemAlloc() or viMemAllocEx() operation, **AND** it has not already been freed, **THEN** the viMemFree() or viMemFreeEx() operation **SHALL** return the corresponding buffer to the device’s memory pool.

**OBSERVATION 6.4.3**

No device is required to have memory that can be shared or managed by the local controller. In this case, a VISA implementation may always return VI\_ERROR\_NSUP\_OPER.

**RULE 6.4.3**

**IF** the offset is currently mapped through the viMapAddress() or viMapAddressEx() operation on the given vi, **THEN** the viMemFree() or viMemFreeEx() operation **SHALL** return VI\_ERROR\_WINDOW\_MAPPED.

6.5 Interface Specific Services

6.5.1 viGpibControlREN(vi, mode)

**Purpose**

Controls the state of the GPIB REN interface line, and optionally the remote/local state of the device.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| Mode | IN | ViUInt16 | Specifies the state of the REN line and optionally the device remote/local state. See the Description section for actual values. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_NCIC | The interface associated with this session is not currently the controller in charge. |
| VI\_ERROR\_NLISTENERS | No listeners condition is detected (both NRFD and NDAC are deasserted). |
| VI\_ERROR\_NSYS\_CNTLR | The interface associated with this session is not the system controller. |
| VI\_ERROR\_INV\_MODE | The value specified by the mode parameter is invalid. |

**Description**

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

Table 6.5.1 Special Values for mode Parameter

|  |  |
| --- | --- |
| **Mode** | **Action Description** |
| VI\_GPIB\_REN\_DEASSERT | Deassert REN line. |
| VI\_GPIB\_REN\_ASSERT | Assert REN line. |
| VI\_GPIB\_REN\_DEASSERT\_GTL | Send the Go To Local command (GTL) to this device and deassert REN line. |
| VI\_GPIB\_REN\_ASSERT\_ADDRESS | Assert REN line and address this device. |
| VI\_GPIB\_REN\_ASSERT\_LLO | Send LLO to any devices that are addressed to listen. |
| VI\_GPIB\_REN\_ASSERT\_ADDRESS\_LLO | Address this device and send it LLO, putting it in RWLS. |
| VI\_GPIB\_REN\_ADDRESS\_GTL | Send the Go To Local command (GTL) to this device. |

**Related Items**

See the INSTR resource description.

**Implementation Requirements**

**RULE 6.5.1**

An INSTR resource implementation of viGpibControlREN() for a GPIB System **SHALL** support all documented modes.

**RULE 6.5.2**

An INTFC resource implementation of viGpibControlREN() for a GPIB System **SHALL** support the modes VI\_GPIB\_REN\_DEASSERT, VI\_GPIB\_REN\_ASSERT, and VI\_GPIB\_REN\_ASSERT\_LLO.

**RULE 6.5.3**

An INSTR resource implementation of viGpibControlREN() for a USB System **SHALL** support all documented modes. The references to addressing the device will have no effect for a USB device.

**RULE 6.5.4**

An INSTR resource implementation of viGpibControlREN() for a USB System **SHALL** return the error VI\_ERROR\_NSUP\_OPER for a USBTMC base-class (non-488) device.

**RULE 6.5.5**

An INSTR resource implementation of viGpibControlREN() for a USB System **SHALL** return the error VI\_ERROR\_NSUP\_OPER for a USBTMC 488-class device that does not implement the optional remote/local state machine.

**RULE 6.5.6**

An INSTR resource implementation of viGpibControlREN() for a TCPIP System **SHALL** support the modes VI\_GPIB\_REN\_DEASSERT\_GTL, VI\_GPIB\_REN\_ASSERT\_ADDRESS, VI\_GPIB\_REN\_ASSERT\_ADRESS\_LLO, and VI\_GPIB\_REN\_ADDRESS\_GTL.

**OBSERVATION 6.5.1**

For a TCPIP device using VXI-11, the modes VI\_GPIB\_REN\_DEASSERT\_GTL and VI\_GPIB\_REN\_ADDRESS\_GTL behave identically, putting the device into local mode. Similarly, the modes VI\_GPIB\_REN\_ASSERT\_ADDRESS and VI\_GPIB\_REN\_ASSERT\_ADRESS\_LLO behave identically, putting the device into remote mode.

**OBSERVATION 6.5.2**

For a TCPIP device using HiSLIP, all modes defined by viGpibControlREN() are supported. However, this specification does not require support for all modes since some do not make sense for TCPIP devices.

6.5.2 viGpibControlATN(vi, mode)

**Purpose**

Controls the state of the GPIB ATN interface line, and optionally the active controller state of the local interface board.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| mode | IN | ViUInt16 | Specifies the state of the ATN line and optionally the local active controller state. See the Description section for actual values. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_NCIC | The interface associated with this session is not currently the controller in charge. |
| VI\_ERROR\_INV\_MODE | The value specified by the mode parameter is invalid. |
| VI\_ERROR\_NSUP\_MODE | The specified mode is not supported by this VISA implementation. |

**Description**

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

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

Table 6.5.2 Special Values for mode Parameter

|  |  |
| --- | --- |
| **Mode** | **Action Description** |
| VI\_GPIB\_ATN\_DEASSERT | Deassert ATN line. |
| VI\_GPIB\_ATN\_ASSERT | Assert ATN line synchronously (in 488 terminology). If a data handshake is in progress, ATN will not be asserted until the handshake is complete. |
| VI\_GPIB\_ATN\_DEASSERT\_HANDSHAKE | Deassert ATN line, and enter shadow handshake mode. The local board will participate in data handshakes as an Acceptor without actually reading the data. |
| VI\_GPIB\_ATN\_ASSERT\_IMMEDIATE | Assert ATN line asynchronously (in 488 terminology). This should generally be used only under error conditions. |

**Related Items**

See the INTFC resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.3 viGpibSendIFC(vi)

**Purpose**

Pulse the interface clear line (IFC) for at least 100 s.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_NSYS\_CNTLR | The interface associated with this session is not the system controller. |

**Description**

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

**Related Items**

See the INTFC resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.4 viGpibCommand(vi, buf, count, retCount)

**Purpose**

Write GPIB command bytes on the bus.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_INV\_SETUP | Unable to start write operation because setup is invalid (due to attributes being set to an inconsistent state). |
| VI\_ERROR\_NCIC | The interface associated with 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. |

**Description**

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

Table 6.5.3 Special Values for retCount Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the number of bytes transferred. |

**Related Items**

See the INTFC resource description.

**Implementation Requirements**

**OBSERVATION 6.5.3**

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

6.5.5 viGpibPassControl(vi, primAddr, secAddr)

**Purpose**

Tell the GPIB device at the specified address to become controller in charge (CIC).

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_NCIC | The interface associated with 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. |

**Description**

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

**Related Items**

See the INTFC resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.6 viVxiCommandQuery(vi, mode, cmd, response)

**Purpose**

Send the device a miscellaneous command or query and/or retrieve the response to a previous query.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| mode | IN | ViUInt16 | Specifies whether to issue a command and/or retrieve a response. See the *Description* section for actual values. |
| cmd | IN | ViUInt32 | The miscellaneous command to send. |
| response | OUT | ViUInt32 | The response retrieved from the device. If the mode specifies just sending a command, this parameter may be VI\_NULL. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Codes** | **Description** |
| VI\_SUCCESS | The operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_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\_OUTP\_PROT\_VIOL | Device reported an output protocol error during transfer. |
| VI\_ERROR\_INP\_PROT\_VIOL | Device reported an input protocol error during transfer. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_RESP\_PENDING | A previous response is still pending, causing a multiple query error. |
| VI\_ERROR\_INV\_MODE | The value specified by the mode parameter is invalid. |

**Description**

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

Table 6.5.4 Special Values for mode Parameter

|  |  |
| --- | --- |
| **Mode** | **Action Description** |
| VI\_VXI\_CMD16 | Send 16-bit Word Serial command. |
| VI\_VXI\_CMD16\_RESP16 | Send 16-bit Word Serial query, get 16-bit response. |
| VI\_VXI\_RESP16 | Get 16-bit response from previous query. |
| VI\_VXI\_CMD32 | Send 32-bit Word Serial command. |
| VI\_VXI\_CMD32\_RESP16 | Send 32-bit Word Serial query, get 16-bit response. |
| VI\_VXI\_CMD32\_RESP32 | Send 32-bit Word Serial query, get 32-bit response. |
| VI\_VXI\_RESP32 | Get 32-bit response from previous query. |

If the mode parameter specifies sending a 16-bit command, the upper half of the cmd parameter is ignored. If the mode parameter specifies just retrieving a response, then the cmd parameter is ignored.

If the mode parameter specifies sending a command only, the response parameter is ignored and may be VI\_NULL. If a response is retrieved but is only a 16-bit value, the upper half of the response parameter will be set to 0.

**Related Items**

See the INSTR resource description.

**Implementation Requirements**

**RULE 6.5.7**

All VISA implementations **SHALL** support all defined mode values for viVxiCommandQuery().

**OBSERVATION 6.5.4**

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

6.5.7 viAssertIntrSignal(vi, mode, statusID)

**Purpose**

Asserts the specified device interrupt or signal.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| mode | IN | ViInt16 | This specifies how to assert the interrupt. See the Description section for actual values. |
| statusID | IN | ViUInt32 | This is the status value to be presented during an interrupt acknowledge cycle. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_BERR | Bus error occurred during transfer. |
| VI\_ERROR\_INTR\_PENDING | An interrupt is still pending from a previous call. |
| VI\_ERROR\_INV\_MODE | The value specified by the mode parameter is invalid. |
| VI\_ERROR\_NSUP\_INTR | The interface cannot generate an interrupt on the requested level or with the requested statusID value. |
| VI\_ERROR\_NSUP\_MODE | The specified mode is not supported by this VISA implementation. |

**Description**

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

Table 6.5.5 Special Values for mode Parameter

|  |  |
| --- | --- |
| **Mode** | **Action Description** |
| VI\_ASSERT\_USE\_ASSIGNED | Use whatever notification method that has been assigned to the local device. |
| VI\_ASSERT\_SIGNAL | Send the notification via a VXI signal. |
| VI\_ASSERT\_IRQ1 - VI\_ASSERT\_IRQ7 | Send the interrupt via the specified VXI/VME IRQ line. This uses the standard VXI/VME ROAK (release on acknowledge) interrupt mechanism rather than the older VME RORA (release on register access) mechanism. |

**Related Items**

See the BACKPLANE and VXI SERVANT resource descriptions.

**Implementation Requirements**

**RULE 6.5.8**

**IF** the mode parameter is VI\_ASSERT\_USE\_ASSIGNED, **AND** vi is a session to a VXI SERVANT resource, **THEN** the operation viAssertIntrSignal() **SHALL** use the mechanism specified in the response of Asynchronous Mode Control command.

**RULE 6.5.9**

**IF** the mode parameter is VI\_ASSERT\_USE\_ASSIGNED, **AND** vi is a session to a BACKPLANE resource, **THEN** the operation viAssertIntrSignal() **SHALL** return the status code VI\_ERROR\_INV\_MODE.

6.5.8 viAssertUtilSignal(vi, line)

**Purpose**

Asserts the specified utility bus signal.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| line | IN | ViUInt16 | Specifies the utility bus signal to assert. This can be the value VI\_UTIL\_ASSERT\_SYSRESET, VI\_UTIL\_ASSERT\_SYSFAIL, or VI\_UTIL\_DEASSERT\_SYSFAIL. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_INV\_LINE | The value specified by the line parameter is invalid. |

**Description**

This operation can be used to assert either the SYSFAIL or SYSRESET utility bus interrupts on the VXIbus backplane. This operation is valid only on VXI BACKPLANE and SERVANT sessions.

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

**Related Items**

See the BACKPLANE and SERVANT resource descriptions.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.9 viMapTrigger(vi, trigSrc, trigDest, mode)

**Purpose**

Map the specified trigger source line to the specified destination line.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| trigSrc | IN | ViInt16 | Source line from which to map. See the *Description* section for actual values. |
| trigDest | IN | ViInt16 | Destination line to which to map. See the *Description* section for actual values. |
| mode | IN | ViUInt16 | Specifies the trigger mapping mode. This should always be VI\_NULL for this version of the specification. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |
| VI\_SUCCESS\_TRIG\_MAPPED | The path from trigSrc to trigDest is already mapped. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_INV\_OBJECT | The given session or object reference is invalid (both are the same value). |
| VI\_ERROR\_NSUP\_OPER | The given vi does not support this operation. |
| VI\_ERROR\_RSRC\_LOCKED | Specified operation could not be performed because the resource identified by vi has been locked for this kind of access. |
| VI\_ERROR\_TMO | Timeout expired before operation completed. |
| VI\_ERROR\_INV\_MODE | The value specified by the mode parameter is invalid. |
| VI\_ERROR\_LINE\_IN\_USE | One of the specified lines (trigSrc or trigDest) is currently in use. |
| VI\_ERROR\_INV\_LINE | One of the specified lines (trigSrc or trigDest) is invalid. |
| VI\_ERROR\_NSUP\_LINE | One of the specified lines (trigSrc or trigDest) is not supported by this VISA implementation. |
| VI\_ERROR\_LINE\_NRESERVED | An attempt was made to use a line that was not reserved. |

**Description**

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

Table 6.5.6 Special Values for trigSrc Parameters

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_TRIG\_TTL0 - VI\_TRIG\_TTL7 | Map the specified TTL trigger line. |
| VI\_TRIG\_ECL0 - VI\_TRIG\_ECL5 | Map the specified VXI ECL trigger line. |
| VI\_TRIG\_STAR\_SLOT1 – VI\_TRIG\_STAR\_SLOT12 | Map the specified STAR input trigger line. |
| VI\_TRIG\_PANEL\_IN | Map the controller’s front panel trigger input line. |

Table 6.5.7 Special Values for trigDest Parameters

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_TRIG\_TTL0 - VI\_TRIG\_TTL7 | Map the specified TTL trigger line. |
| VI\_TRIG\_ECL0 - VI\_TRIG\_ECL5 | Map the specified VXI ECL trigger line. |
| VI\_TRIG\_STAR\_VXI0 – VI\_TRIG\_STAR\_VXI2 | Map the specified VXI STAR trigger output line. |
| VI\_TRIG\_PANEL\_OUT | Map the controller’s front panel trigger output line. |

If this operation is called multiple times on the same BACKPLANE resource with the same source trigger line and different destination trigger lines, the result should be that when the source trigger line is asserted, all of the specified destination trigger lines should also be asserted. If this operation is called multiple times on the same BACKPLANE resource with different source trigger lines and the same destination trigger line, the result should be that when any of the specified source trigger lines is asserted, the destination trigger line should also be asserted. However, mapping a trigger line (as either source or destination) multiple times requires special hardware capabilities and is not guaranteed to be implemented.

**Related Items**

See the BACKPLANE resource description.

**Implementation Requirements**

# RULE 6.5.10

**IF** a VISA implementation does not support mapping the same trigger line multiple times, **AND** either trigSrc or trigDest specifies a line that is already mapped, **THEN** viMapTrigger() **SHALL** return the status code VI\_ERROR\_LINE\_IN\_USE.

# RULE 6.5.11

**IF** a path already exists from trigSrc to trigDest, **THEN** viMapTrigger() **SHALL NOT** create a new hardware trigger mapping and **SHALL** return the status code VI\_SUCCESS\_TRIG\_MAPPED.

# RULE 6.5.12

A PXI implementation of viMapTrigger() **SHALL** use the current value of VI\_ATTR\_PXI\_SRC\_TRIG\_BUS to qualify trigSrc **AND** it **SHALL** use the current value of VI\_ATTR\_PXI\_DEST\_TRIG\_BUS to qualify trigDest.

# PERMISSION 6.5.1

A vendor implementation of viMapTrigger() **MAY** support mapping between trigger lines that do not support a direct path but need intermediate lines to be used for the map.

# RULE 6.5.13

A successful call to viMapTrigger() for PXI **SHALL** result in a state where the destination trigger line/bus pair as well as all intermediate trigger line/bus pairs that are required to create the path are reserved. **IF** all of the necessary line/bus pairs have already been reserved using the current VISA resource and vendor VISA implementation **AND** the line/bus pairs are not part of an existing map, **THEN** viMapTrigger() **SHALL** reuse the reservation.

# PERMISSION 6.5.2

If the destination line/bus and intermediate line/bus pairs were not reserved before the call to viMapTrigger() for PXI, a vendor implementation of this function **MAY** implicitly reserve such line/bus pairs before attempting to create the path.

# RULE 6.5.14

A successful call to viMapTrigger() for PXI **SHALL NOT** implicitly cause the source bus segment and source trigger line to be reserved.

# OBSERVATION 6.5.5

Mapping one trigger line to another modifies the state of hardware. As such, the effect continues beyond the scope of the VISA session that mapped it, even if that VISA session is closed.

6.5.10 viUnmapTrigger(vi, trigSrc, trigDest)

**Purpose**

Undo a previous map from the specified trigger source line to the specified destination line.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| Vi | IN | ViSession | Unique logical identifier to a session. |
| trigSrc | IN | ViInt16 | Source line used in previous map. See the *Description* section for actual values. |
| trigDest | IN | ViInt16 | Destination line used in previous map. See the *Description* section for actual values. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_LINE | One of the specified lines (trigSrc or trigDest) is invalid. |
| VI\_ERROR\_TRIG\_NMAPPED | The path from trigSrc to trigDest is not currently mapped. |
| VI\_ERROR\_NSUP\_LINE | One of the specified lines (trigSrc or trigDest) is not supported by this VISA implementation. |

**Description**

This operation can be used to undo a previous mapping of one trigger line to another. This operation is valid only on BACKPLANE (mainframe) sessions.

Table 6.5.7 Special Values for trigSrc Parameters

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_TRIG\_TTL0 - VI\_TRIG\_TTL7 | Unmap the specified TTL trigger line. |
| VI\_TRIG\_ECL0 - VI\_TRIG\_ECL5 | Unmap the specified VXI ECL trigger line. |
| VI\_TRIG\_STAR\_SLOT0 – VI\_TRIG\_STAR\_SLOT12 | Unmap the specified STAR input trigger line. |
| VI\_TRIG\_PANEL\_IN | Unmap the controller’s front panel trigger input line. |

Table 6.5.8 Special Values for trigDest Parameters

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_TRIG\_TTL0 - VI\_TRIG\_TTL7 | Unmap the specified TTL trigger line. |
| VI\_TRIG\_ECL0 - VI\_TRIG\_ECL5 | Unmap the specified VXI ECL trigger line. |
| VI\_TRIG\_STAR\_VXI0 – VI\_TRIG\_STAR\_VXI2 | Unmap the specified VXI STAR trigger output line. |
| VI\_TRIG\_PANEL\_OUT | Unmap the controller’s front panel trigger output line. |
| VI\_TRIG\_ALL | Unmap all trigger lines to which trigSrc is currently connected. |

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

**Related Items**

See the BACKPLANE resource description.

**Implementation Requirements**

# RULE 6.5.15

**IF** the viMapTrigger() function for PXI implicitly reserved one or more line/bus pairs when mapping from trigSrc to trigDest, **THEN** a successful call to viUnmapTrigger() **SHALL** implicitly unreserve those line/bus pairs.

**RULE 6.5.16**

**IF** the viMapTrigger() function for PXI did not implicitly reserve any line/bus pairs when mapping from trigSrc to trigDest, **THEN** viUnmapTrigger() **SHALL NOT** change the reservation state of those line/bus pairs.

6.5.11 viUsbControlOut (vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf)

**Purpose**

Send arbitrary data to the USB device on the control port.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| bmRequestType | IN | ViInt16 | Bitmap field for defining the USB control port request. The bitmap fields are as defined by the USB specification. The direction bit must be host-to-device. |
| bRequest | IN | ViInt16 | Request ID for this transfer. The meaning of this value depends on bmRequestType. |
| wValue | IN | ViUInt16 | Request value for this transfer. |
| wIndex | IN | ViUInt16 | Specifies the interface or endpoint index number, depending on bmRequestType. |
| wLength | IN | ViUInt16 | Length of the data in bytes to send to the device during the Data stage. If this value is 0, then buf is ignored. |
| buf | IN | ViBuf | Actual data to send to the device during the Data stage. If wLength is 0, then this parameter is ignored. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_MASK | The value in bmRequestType does not have the direction bit set to the correct value. |
| VI\_ERROR\_IO | Could not perform operation because of I/O error. |
| VI\_ERROR\_INV\_PARAMETER | The high byte of bmRequestType or bRequest is not zero. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

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

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

**Related Items**

See the USB INSTR resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.12 viUsbControlIn (vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf, retCnt)

**Purpose**

Request arbitrary data from the USB device on the control port.

**Parameters**

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

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_MASK | The value in bmRequestType does not have the direction bit set to the correct value. |
| VI\_ERROR\_IO | Could not perform operation because of I/O error. |
| VI\_ERROR\_INV\_PARAMETER | The high byte of bmRequestType or bRequest is not zero. |
| VI\_ERROR\_CONN\_LOST | The I/O connection for the given session has been lost. |

**Description**

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

Table 6.5.9 Special Values for retCnt Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the actual number of bytes read from the control pipe. |

**Related Items**

See the USB INSTR resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

6.5.13 viPxiReserveTriggers (vi, cnt, trigBuses,  
trigLines, failureIndex)

**Purpose**

Reserves multiple trigger lines that the caller can then map and/or assert.

**Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Direction** | **Type** | **Description** |
| vi | IN | ViSession | Unique logical identifier to a session. |
| cnt | IN | ViInt16 | Number of trigger bus/line pairs to follow. |
| trigBuses | IN | ViAInt16 | Array of trigger buses. The size of this array is specified in cnt. |
| trigLines | IN | ViAInt16 | Array of trigger lines. The size of this array is specified in cnt. |
| failureIndex | OUT | ViInt16 | Specifies the 0-based index of the first trigger bus/line pair that could not be reserved, if this function returns an error code directly related to reserving triggers. On success, this output parameter contains the value -1. For any other status code returned, the value of this output parameter is undefined and should not be used. |

**Return Values**

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

|  |  |
| --- | --- |
| **Completion Code** | **Description** |
| VI\_SUCCESS | Operation completed successfully. |

|  |  |
| --- | --- |
| **Error Codes** | **Description** |
| VI\_ERROR\_INV\_SESSION VI\_ERROR\_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\_LENGTH | Invalid count specified. |
| VI\_ERROR\_IO | Could not perform operation because of I/O error. |
| VI\_ERROR\_LINE\_IN\_USE | One of the specified lines is currently in use. |
| VI\_ERROR\_INV\_LINE | One of the specified lines is invalid. |
| VI\_ERROR\_NSUP\_LINE | One of the specified lines is not supported by this VISA implementation. |

**Description**

For a PXI BACKPLANE resource, viPxiReserveTriggers() will reserve multiple triggers for later use by the client, such as for assertion and/or mapping. This operation is intended to be atomic, such that if it is not possible to simultaneously reserve all the requested bus/line pairs, then none of the bus/line pairs will be reserved.

Table 6.5.9 Special Values for failureIndex Parameter

|  |  |
| --- | --- |
| **Value** | **Action Description** |
| VI\_NULL | Do not return the index of the first failure. |

**Related Items**

See the PXI BACKPLANE resource description.

**Implementation Requirements**

There are no additional implementation requirements other than those specified above.

Appendix A Required Attributes

This appendix lists the required attributes along with the range and default value of every resource described in this document.

The set of required attributes varies from interface to interface, and the range and default values for individual attributes may also vary from interface to interface. The set of required attributes for a write operation for the VXI interface, for example, is different from that of a write operation for the GPIB interface. In this appendix, such resources will have several tables of required attributes, one for each type of interface that the resource must be capable of supporting.

A.1 Required Attribute Tables

Resource Template Attributes

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_TYPE | N/A | N/A |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh VI\_TMO\_INFINITE | 2000 |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |

(continues)

INSTR Resource Attributes (Generic) (Continued)

| **Symbolic Name** | **Range** | **Default** |
| --- | --- | --- |
| VI\_ATTR\_INTF\_TYPE | N/A | N/A |
| VI\_ATTR\_TRIG\_ID | VI\_TRIG\_SW; VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_VXI0 to  VI\_TRIG\_STAR\_VXI2;  VI\_TRIG\_STAR\_INSTR | VI\_TRIG\_SW |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |

INSTR Resource Attributes (Message Based)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_IO\_PROT | VI\_PROT\_NORMAL VI\_PROT\_FDC VI\_PROT\_HS488 VI\_PROT\_4882\_STRS VI\_PROT\_USBTMC\_VENDOR | VI\_PROT\_NORMAL |
| VI\_ATTR\_SEND\_END\_EN | VI\_TRUE VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_SUPPRESS\_END\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_TERMCHAR | 0 to FFh | 0Ah (linefeed) |
| VI\_ATTR\_TERMCHAR\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_FILE\_APPEND\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |

INSTR Resource Attributes (GPIB and GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | 0 to 30 | N/A |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | 0 to 31, VI\_NO\_SEC\_ADDR | N/A |
| VI\_ATTR\_GPIB\_READDR\_EN | VI\_TRUE VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_GPIB\_UNADDR\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_GPIB\_REN\_STATE | VI\_STATE\_ASSERTED  VI\_STATE\_UNASSERTED  VI\_STATE\_UNKNOWN | N/A |

INSTR Resource Attributes (VXI, GPIB-VXI, and PXI Specific**)**

| **Symbolic Name** | **Range** | **Default** |
| --- | --- | --- |
| VI\_ATTR\_Slot | N/A | N/A |
| VI\_ATTR\_WIN\_ACCESS | VI\_TRUE VI\_FALSE | N/A |

INSTR Resource Attributes (VXI and GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_Slot | 0 to 7 | N/A |
| VI\_ATTR\_MEM\_Base\_32  VI\_ATTR\_MEM\_Base\_64 | N/A | N/A |
| VI\_ATTR\_MEM\_Size\_32  VI\_ATTR\_MEM\_Size\_64 | N/A | N/A |
| VI\_ATTR\_MEM\_Space | VI\_NMAPPED VI\_A24\_SPACE VI\_DEREF\_ADDR | VI\_NMAPPED |
| VI\_ATTR\_SRC\_INCREMENT | 0 to 1 | 1 |
| VI\_ATTR\_DEST\_INCREMENT | 0 to 1 | 1 |

INSTR Resource Attributes (VXI and GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_FDC\_CHNL | 0 to 7 | N/A |
| VI\_ATTR\_FDC\_MODE | VI\_FDC\_NORMAL VI\_FDC\_STREAM | VI\_DATA\_PRIV |
| VI\_ATTR\_FDC\_USE\_PAIR | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_VXI\_DEV\_CLASS | VI\_TRUE VI\_FALSE | N/A |
| VI\_ATTR\_VXI\_TRIG\_SUPPORT | N/A | N/A |

**INSTR Resource Attributes (GPIB-VXI Specific)**

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_PARENT\_NUM | 0 to FFFFh | N/A |

INSTR Resource Attributes (ASRL Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_ASRL\_AVAIL\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_ASRL\_BAUD | N/A | N/A |
| VI\_ATTR\_ASRL\_DATA\_BITS | N/A | 8 |

INSTR Resource Attributes (TCPIP Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_TCPIP\_ADDR | N/A | N/A |
| VI\_ATTR\_TCPIP\_HOSTNAME | N/A | N/A |
| VI\_ATTR\_TCPIP\_DEVICE\_NAME | N/A | N/A |
| VI\_ATTR\_TCPIP\_IS\_HISLIP | VI\_TRUE  VI\_FALSE | N/A |

INSTR Resource Attributes (HiSLIP Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_TCPIP\_HISLIP\_VERSION | 0h to FFFFFFFFh | N/A |
| VI\_ATTR\_TCPIP\_HISLIP\_MAX\_MESSAGE\_KB | 0h to FFFFFFFFh | 1024 |
| VI\_ATTR\_TCPIP\_HISLIP\_OVERLAP\_EN | VI\_TRUE  VI\_FALSE | Preference returned by device. |
| VI\_ATTR\_TCPIP\_PORT | 0 to FFFFh | 4880 |
| VI\_ATTR\_TCPIP\_NODELAY | VI\_TRUE  VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_TCPIP\_KEEPALIVE | VI\_TRUE  VI\_FALSE | VI\_FALSE |

INSTR Resource Attributes (VXI, GPIB-VXI, USB, and PXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_MANF\_ID | 0 to FFFFh | 0 |
| VI\_ATTR\_MODEL\_CODE | 0 to FFFFh | 0 |
| VI\_ATTR\_MANF\_NAME | N/A | N/A |
| VI\_ATTR\_MODEL\_NAME | N/A | N/A |

INSTR Resource Attributes (VXI, GPIB-VXI, and USB Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_4882\_COMPLIANT | VI\_TRUE  VI\_FALSE | N/A |

INSTR Resource Attributes (USB Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_USB\_SERIAL\_NUM | N/A | N/A |
| VI\_ATTR\_USB\_INTFC\_NUM | 0 to 254 | 0 |
| VI\_ATTR\_USB\_MAX\_INTR\_SIZE | 0 to FFFFh | N/A |
| VI\_ATTR\_USB\_PROTOCOL | 0 to 255 | N/A |

INSTR Resource Attributes (PXI Specific)

|  |  |  |
| --- | --- | --- |
| Symbolic Name | Range | Default |
| VI\_ATTR\_PXI\_DEV\_NUM | 0 to 31 | N/A |
| VI\_ATTR\_PXI\_FUNC\_NUM | 0 to 7 | N/A |
| VI\_ATTR\_PXI\_BUS\_NUM | 0 to 255 | N/A |
| VI\_ATTR\_PXI\_CHASSIS | 0 to 255 VI\_UNKNOWN\_CHASSIS | N/A |

(continues)

**INSTR Resource Attributes (PXI Specific) (Continued)**

|  |  |  |
| --- | --- | --- |
| Symbolic Name | Range | Default |
| VI\_ATTR\_PXI\_SLOTPATH | N/A | N/A |
| VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT | 0 to 32767 VI\_UNKNOWN\_SLOT | N/A |
| VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT | 0 to 32767 VI\_UNKNOWN\_SLOT | N/A |
| VI\_ATTR\_PXI\_TRIG\_BUS | 0 to 32767 VI\_UNKNOWN\_TRIG | N/A |
| VI\_ATTR\_PXI\_STAR\_TRIG\_BUS | 0 to 32767 VI\_UNKNOWN\_TRIG | N/A |
| VI\_ATTR\_PXI\_STAR\_TRIG\_LINE | 0 to 32767  VI\_UNKNOWN\_TRIG | N/A |
| VI\_ATTR\_PXI\_MEM\_TYPE\_BARn (where *n* is 0,1,2,3,4,5) | VI\_PXI\_ADDR\_MEM, VI\_PXI\_ADDR\_IO, VI\_PXI\_ADDR\_NONE | N/A |
| VI\_ATTR\_PXI\_MEM\_BASE\_BARn \_32  VI\_ATTR\_PXI\_MEM\_BASE\_BARn \_64  (where *n* is 0,1,2,3,4,5) | N/A | N/A |
| VI\_ATTR\_PXI\_MEM\_SIZE\_BARn \_32 VI\_ATTR\_PXI\_MEM\_SIZE\_BARn \_64 (where *n* is 0,1,2,3,4,5) | N/A | N/A |

MEMACC Resource Attributes (Generic)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_INTF\_TYPE | VI\_INTF\_VXI  VI\_INTF\_GPIB\_VXI | N/A |
| VI\_ATTR\_INTF\_INST\_NAME | N/A | N/A |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh VI\_TMO\_INFINITE | 2000 |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_TRUE VI\_FALSE | N/A |

MEMACC Resource Attributes (VXI, GPIB-VXI, and PXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_SRC\_INCREMENT | 0 to 1 | 1 |
| VI\_ATTR\_DEST\_INCREMENT | 0 to 1 | 1 |
| VI\_ATTR\_WIN\_BASE\_ADDR\_32  VI\_ATTR\_WIN\_BASE\_ADDR\_64 | N/A | N/A |
| VI\_ATTR\_WIN\_SIZE\_32  VI\_ATTR\_WIN\_SIZE\_64 | N/A | N/A |
| VI\_ATTR\_WIN\_ACCESS | VI\_NMAPPED VI\_USE\_OPERS VI\_DEREF\_ADDR | VI\_NMAPPED |

MEMACC Resource Attributes (VXI and GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_VXI\_LA | 0 to 255 | N/A |
| VI\_ATTR\_SRC\_BYTE\_ORDER | VI\_BIG\_ENDIAN VI\_LITTLE\_ENDIAN | VI\_BIG\_ENDIAN |
| VI\_ATTR\_DEST\_BYTE\_ORDER | VI\_BIG\_ENDIAN VI\_LITTLE\_ENDIAN | VI\_BIG\_ENDIAN |
| VI\_ATTR\_WIN\_BYTE\_ORDER | VI\_BIG\_ENDIAN VI\_LITTLE\_ENDIAN | VI\_BIG\_ENDIAN |
| VI\_ATTR\_SRC\_ACCESS\_PRIV | VI\_DATA\_NPRIV VI\_DATA\_PRIV VI\_PROG\_NPRIV VI\_PROG\_PRIV VI\_BLCK\_NPRIV VI\_BLCK\_PRIV VI\_D64\_NPRIV VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 | VI\_DATA\_PRIV |
| VI\_ATTR\_DEST\_ACCESS\_PRIV | VI\_DATA\_NPRIV VI\_DATA\_PRIV VI\_PROG\_NPRIV VI\_PROG\_PRIV VI\_BLCK\_NPRIV VI\_BLCK\_PRIV VI\_D64\_NPRIV VI\_D64\_PRIV  VI\_D64\_2EVME  VI\_D64\_SST160  VI\_D64\_SST267  VI\_D64\_SST320 | VI\_DATA\_PRIV |
| VI\_ATTR\_WIN\_ACCESS\_PRIV | VI\_DATA\_NPRIV VI\_DATA\_PRIV VI\_PROG\_NPRIV VI\_PROG\_PRIV VI\_BLCK\_NPRIV VI\_BLCK\_PRIV | VI\_DATA\_PRIV |

MEMACC Resource Attributes (GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_PARENT\_NUM | 0 to FFFFh | N/A |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | 0 to 30 | N/A |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | 0 to 31, VI\_NO\_SEC\_ADDR | N/A |

INTFC Resource Attributes (Generic)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_INTF\_TYPE | VI\_INTF\_GPIB | VI\_INTF\_GPIB |
| VI\_ATTR\_INTF\_INST\_NAME | N/A | N/A |
| VI\_ATTR\_SEND\_END\_EN | VI\_TRUE VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_TERMCHAR | 0 to FFh | 0Ah (linefeed) |
| VI\_ATTR\_TERMCHAR\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh | 2000 |
| VI\_ATTR\_DEV\_STATUS\_BYTE | 0 to FFh | N/A |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | VI\_FLUSH\_ON\_ACCESS VI\_FLUSH\_WHEN\_FULL | VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_TRUE VI\_FALSE | N/A |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | VI\_FLUSH\_ON\_ACCESS VI\_FLUSH\_DISABLE | VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |

INTFC Resource Attributes (GPIB Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | 0 to 30 | N/A |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | 0 to 31 VI\_NO\_SEC\_ADDR | VI\_NO\_SEC\_ADDR |
| VI\_ATTR\_GPIB\_REN\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_GPIB\_ATN\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_GPIB\_NDAC\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_GPIB\_SRQ\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_GPIB\_CIC\_STATE | VI\_TRUE VI\_FALSE | N/A |
| VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE | VI\_TRUE VI\_FALSE | N/A |
| VI\_ATTR\_GPIB\_HS488\_CBL\_LEN | 1 to 15 VI\_GPIB\_HS488\_DISABLED VI\_GPIB\_HS488\_NIMPL | N/A |

BACKPLANE Resource Attributes (Generic)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_INTF\_TYPE | VI\_INTF\_VXI  VI\_INTF\_GPIB\_VXI | N/A |
| VI\_ATTR\_INTF\_INST\_NAME | N/A | N/A |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh VI\_TMO\_INFINITE | 2000 |

BACKPLANE Resource Attributes (VXI and GPIB-VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_TRIG\_ID | VI\_TRIG\_SW; VI\_TRIG\_TTL0 to VI\_TRIG\_TTL7; VI\_TRIG\_ECL0 to VI\_TRIG\_ECL5;  VI\_TRIG\_STAR\_SLOT1 to  VI\_TRIG\_STAR\_SLOT12;  VI\_TRIG\_STAR\_VXI0 to  VI\_TRIG\_STAR\_VXI2;  VI\_TRIG\_PANEL\_IN;  VI\_TRIG\_PANEL\_OUT | N/A |
| VI\_ATTR\_MAINFRAME\_LA | 0 to 255 VI\_UNKNOWN\_LA | N/A |
| VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_VXI\_VME\_INTR\_STATUS | N/A | N/A |
| VI\_ATTR\_VXI\_TRIG\_STATUS | N/A | N/A |
| VI\_ATTR\_VXI\_TRIG\_SUPPORT | N/A | N/A |

SERVANT Resource Attributes (Generic)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_INTF\_TYPE | VI\_INTF\_VXI VI\_INTF\_GPIB VI\_INTF\_TCPIP | N/A |
| VI\_ATTR\_INTF\_INST\_NAME | N/A | N/A |
| VI\_ATTR\_SEND\_END\_EN | VI\_TRUE VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_TERMCHAR | 0 to FFh | 0Ah (linefeed) |
| VI\_ATTR\_TERMCHAR\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh | 2000 |
| VI\_ATTR\_DEV\_STATUS\_BYTE | 0 to FFh | N/A |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | VI\_FLUSH\_ON\_ACCESS VI\_FLUSH\_WHEN\_FULL | VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_TRUE VI\_FALSE | N/A |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | VI\_FLUSH\_ON\_ACCESS VI\_FLUSH\_DIABLE | VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | VI\_TRUE VI\_FALSE | VI\_FALSE |

SERVANT Resource Attributes (GPIB Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | 0 to 30 | N/A |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | 0 to 31, VI\_NO\_SEC\_ADDR | VI\_NO\_SEC\_ADDR |
| VI\_ATTR\_GPIB\_REN\_STATE | VI\_STATE\_ASSERTED VI\_STATE\_UNASSERTED VI\_STATE\_UNKNOWN | N/A |
| VI\_ATTR\_GPIB\_ADDR\_STATE | VI\_GIPB\_UNADDRESSED VI\_GPIB\_TALKER VI\_GPIB\_LISTENER | N/A |

SERVANT Resource Attributes (VXI Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_VXI\_LA | 0 to 511 | N/A |
| VI\_ATTR\_CMDR\_LA | 0 to 255 VI\_UNKNOWN\_LA | N/A |

SERVANT Resource Attributes (TCPIP Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_TCPIP\_DEVICE\_NAME | N/A | N/A |

SOCKET Resource Attributes (Generic)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_INTF\_NUM | 0 to FFFFh | 0 |
| VI\_ATTR\_INTF\_TYPE | VI\_INTF\_TCPIP | VI\_INTF\_TCPIP |
| VI\_ATTR\_INTF\_INST\_NAME | N/A | N/A |
| VI\_ATTR\_SEND\_END\_EN | VI\_TRUE  VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_TERMCHAR | 0 to FFh | 0Ah (linefeed) |
| VI\_ATTR\_TERMCHAR\_EN | VI\_TRUE  VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_TMO\_VALUE | VI\_TMO\_IMMEDIATE 1 to FFFFFFFEh VI\_TMO\_INFINITE | 2000 |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | VI\_FLUSH\_ACCESS VI\_FLUSH\_WHEN\_FULL | VI\_FLUSH\_WHEN\_FULL |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_TRUE  VI\_FALSE | VI\_FALSE |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | VI\_FLUSH\_ON\_ACCESS VI\_FLUSH\_DISABLE | VI\_FLUSH\_DISABLE |
| VI\_ATTR\_FILE\_APPEND\_EN | VI\_TRUE  VI\_FALSE | VI\_FALSE |

SOCKET Resource Attributes (TCPIP Specific)

|  |  |  |
| --- | --- | --- |
| **Symbolic Name** | **Range** | **Default** |
| VI\_ATTR\_TCPIP\_ADDR | N/A | N/A |
| VI\_ATTR\_TCPIP\_HOSTNAME | N/A | N/A |
| VI\_ATTR\_TCPIP\_PORT | 0 to FFFFh | N/A |
| VI\_ATTR\_TCPIP\_NODELAY | VI\_TRUE  VI\_FALSE | VI\_TRUE |
| VI\_ATTR\_TCPIP\_KEEPALIVE | VI\_TRUE  VI\_FALSE | VI\_FALSE |

Appendix B Resource Summary Information

B.1 Summary of Attributes

**VISA Resource Template**

(These attributes are based on the VISA Resource Template and are available to all other resources.)

VI\_ATTR\_MAX\_QUEUE\_LENGTH

VI\_ATTR\_RM\_SESSION

VI\_ATTR\_RSRC\_IMPL\_VERSION

VI\_ATTR\_RSRC\_LOCK\_STATE

VI\_ATTR\_RSRC\_MANF\_ID

VI\_ATTR\_RSRC\_MANF\_NAME

VI\_ATTR\_RSRC\_NAME

VI\_ATTR\_RSRC\_SPEC\_VERSION

VI\_ATTR\_USER\_DATA

**INSTR Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_ASRL\_AVAIL\_NUM | VI\_ATTR\_ASRL\_BAUD |
| VI\_ATTR\_ASRL\_CTS\_STATE | VI\_ATTR\_ASRL\_DATA\_BITS |
| VI\_ATTR\_ASRL\_DCD\_STATE | VI\_ATTR\_ASRL\_DSR\_STATE |
| VI\_ATTR\_ASRL\_DTR\_STATE | VI\_ATTR\_ASRL\_END\_IN |
| VI\_ATTR\_ASRL\_END\_OUT | VI\_ATTR\_ASRL\_FLOW\_CNTRL |
| VI\_ATTR\_ASRL\_PARITY | VI\_ATTR\_ASRL\_REPLACE\_CHAR |
| VI\_ATTR\_ASRL\_RI\_STATE | VI\_ATTR\_ASRL\_RTS\_STATE |
| VI\_ATTR\_ASRL\_STOP\_BITS | VI\_ATTR\_ASRL\_XON\_CHAR |
| VI\_ATTR\_ASRL\_XOFF\_CHAR | VI\_ATTR\_GPIB\_REN\_STATE |
| VI\_ATTR\_CMDR\_LA | VI\_ATTR\_DEST\_ACCESS\_PRIV |
| VI\_ATTR\_DEST\_BYTE\_ORDER | VI\_ATTR\_DEST\_INCREMENT |
| VI\_ATTR\_FDC\_CHNL | VI\_ATTR\_FDC\_GEN\_SIGNAL\_EN |
| VI\_ATTR\_FDC\_MODE | VI\_ATTR\_FDC\_USE\_PAIR |
| VI\_ATTR\_GPIB\_PRIMARY\_ADDR | VI\_ATTR\_GPIB\_READDR\_EN |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | VI\_ATTR\_GPIB\_UNADDR\_EN |
| VI\_ATTR\_IMMEDIATE\_SERV | VI\_ATTR\_INTF\_INST\_NAME |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_INTF\_PARENT\_NUM |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_IO\_PROT |
| VI\_ATTR\_MAINFRAME\_LA | VI\_ATTR\_ManF\_Id |
| VI\_ATTR\_MEM\_Base\_32 | VI\_ATTR\_MEM\_Size\_32 |
| VI\_ATTR\_MEM\_Space | VI\_ATTR\_Model\_Code |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | VI\_ATTR\_SEND\_END\_EN |
| VI\_ATTR\_Slot | VI\_ATTR\_SRC\_ACCESS\_PRIV |
| VI\_ATTR\_SRC\_BYTE\_ORDER | VI\_ATTR\_SRC\_INCREMENT |
| VI\_ATTR\_SUPPRESS\_END\_EN | VI\_ATTR\_TERMCHAR |
| VI\_ATTR\_TERMCHAR\_EN | VI\_ATTR\_TMO\_VALUE |
| VI\_ATTR\_TRIG\_ID | VI\_ATTR\_VXI\_LA |
| VI\_ATTR\_WIN\_ACCESS | VI\_ATTR\_WIN\_ACCESS\_PRIV |
| VI\_ATTR\_WIN\_BASE\_ADDR\_32 | VI\_ATTR\_WIN\_BYTE\_ORDER |
| VI\_ATTR\_WIN\_SIZE\_32 | VI\_ATTR\_WR\_BUF\_OPER\_MODE |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_ATTR\_VXI\_TRIG\_SUPPORT |
| VI\_ATTR\_VXI\_DEV\_CLASS | VI\_ATTR\_TCPIP\_ADDR |
| VI\_ATTR\_MANF\_NAME | VI\_ATTR\_TCPIP\_HOSTNAME |
| VI\_ATTR\_FILE\_APPEND\_EN | VI\_ATTR\_TCPIP\_PORT |
| VI\_ATTR\_MODEL\_NAME | VI\_ATTR\_4882\_COMPLIANT |
| VI\_ATTR\_USB\_SERIAL\_NUM | VI\_ATTR\_USB\_INTFC\_NUM |
| VI\_ATTR\_USB\_MAX\_INTR\_SIZE | VI\_ATTR\_USB\_PROTOCOL |
| VI\_ATTR\_RD\_BUF\_SIZE | VI\_ATTR\_WR\_BUF\_SIZE |
| VI\_ATTR\_PXI\_BUS\_NUM | VI\_ATTR\_PXI\_CHASSIS |
| VI\_ATTR\_PXI\_DEV\_NUM | VI\_ATTR\_PXI\_FUNC\_NUM |
| VI\_ATTR\_PXI\_MEM\_BASE\_BAR0 \_32– VI\_ATTR\_PXI\_MEM\_BASE\_BAR5\_32 | VI\_ATTR\_PXI\_MEM\_SIZE\_BAR0\_32 – VI\_ATTR\_PXI\_MEM\_SIZE\_BAR5\_32 |
| VI\_ATTR\_PXI\_MEM\_TYPE\_BAR0 – VI\_ATTR\_PXI\_MEM\_TYPE\_BAR5 | VI\_ATTR\_PXI\_MEM\_SIZE\_BAR0\_64 - VI\_ATTR\_PXI\_MEM\_SIZE\_BAR5\_64 |
| VI\_ATTR\_PXI\_SLOT\_LBUS\_LEFT  VI\_ATTR\_PXI\_SLOT\_LBUS\_RIGHT | VI\_ATTR\_PXI\_MEM\_BASE\_BAR0\_64 - |
| VI\_ATTR\_PXI\_STAR\_TRIG\_BUS | VI\_ATTR\_PXI\_MEM\_BASE\_BAR5\_64 |
| VI\_ATTR\_PXI\_TRIG\_BUS | VI\_ATTR\_PXI\_SLOTPATH |
| VI\_ATTR\_WIN\_SIZE\_64 | VI\_ATTR\_PXI\_STAR\_TRIG\_LINE |
| VI\_ATTR\_MEM\_SIZE\_64 | VI\_ATTR\_WIN\_BASE\_ADDR\_64 |
| VI\_ATTR\_TCPIP\_HISLIP\_VERSION | VI\_ATTR\_MEM\_BASE\_64 |
| VI\_ATTR\_TCPIP\_HISLIP\_MAX\_MESSAGE\_KB  VI\_ATTR\_PXI\_ALLOW\_WRITE\_COMBINe | VI\_ATTR\_TCPIP\_HISLIP\_OVERLAP\_EN |

**MEMACC Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_DEST\_ACCESS\_PRIV | VI\_ATTR\_DEST\_BYTE\_ORDER |
| VI\_ATTR\_DEST\_INCREMENT | VI\_ATTR\_GPIB\_PRIMARY\_ADDR |
| VI\_ATTR\_GPIB\_SECONDARY\_ADDR | VI\_ATTR\_INTF\_INST\_NAME |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_INTF\_PARENT\_NUM |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_SRC\_ACCESS\_PRIV |
| VI\_ATTR\_SRC\_BYTE\_ORDER | VI\_ATTR\_SRC\_INCREMENT |
| VI\_ATTR\_TMO\_VALUE | VI\_ATTR\_VXI\_LA |
| VI\_ATTR\_WIN\_ACCESS | VI\_ATTR\_WIN\_ACCESS\_PRIV |
| VI\_ATTR\_WIN\_BASE\_ADDR\_32 | VI\_ATTR\_WIN\_BYTE\_ORDER |
| VI\_ATTR\_WIN\_SIZE\_32 | VI\_ATTR\_DMA\_ALLOW\_EN |
| VI\_ATTR\_WIN\_BASE\_ADDR\_64 | VI\_ATTR\_WIN\_SIZE\_64 |

**INTFC Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_FILE\_APPEND\_EN |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_GPIB\_PRIMARY\_ADDR |
| VI\_ATTR\_INTF\_INST\_NAME | VI\_ATTR\_GPIB\_SECONDARY\_ADDR |
| VI\_ATTR\_SEND\_END\_EN | VI\_ATTR\_GPIB\_REN\_STATE |
| VI\_ATTR\_TERMCHAR | VI\_ATTR\_GPIB\_ATN\_STATE |
| VI\_ATTR\_TERMCHAR\_EN | VI\_ATTR\_GPIB\_NDAC\_STATE |
| VI\_ATTR\_TMO\_VALUE | VI\_ATTR\_GPIB\_SRQ\_STATE |
| VI\_ATTR\_DEV\_STATUS\_BYTE | VI\_ATTR\_GPIB\_CIC\_STATE |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | VI\_ATTR\_GPIB\_SYS\_CNTRL\_STATE |
| VI\_ATTR\_DMA\_ALLOW\_EN | VI\_ATTR\_GPIB\_HS488\_CBL\_LEN |
| VI\_ATTR\_RD\_BUF\_OPER\_MODE | VI\_ATTR\_GPIB\_ADDR\_STATE |
| VI\_ATTR\_RD\_BUF\_SIZE | VI\_ATTR\_WR\_BUF\_SIZE |

**BACKPLANE Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_MAINFRAME\_LA |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_VXI\_VME\_SYSFAIL\_STATE |
| VI\_ATTR\_INTF\_INST\_NAME | VI\_ATTR\_VXI\_VME\_INTR\_STATUS |
| VI\_ATTR\_TMO\_VALUE | VI\_ATTR\_VXI\_TRIG\_STATUS |
| VI\_ATTR\_TRIG\_ID | VI\_ATTR\_GPIB\_PRIMARY\_ADDR |
| VI\_ATTR\_VXI\_TRIG\_SUPPORT  VI\_ATTR\_PXI\_SRC\_TRIG\_BUS  VI\_ATTR\_PXI\_DEST\_TRIG\_BUS | VI\_ATTR\_GPIB\_SECONDARY\_ADDR  VI\_ATTR\_INTF\_PARENT\_NUM |
|  |  |

**SERVANT Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_DMA\_ALLOW\_EN |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_RD\_BUF\_OPER\_MODE |
| VI\_ATTR\_INTF\_INST\_NAME | VI\_ATTR\_FILE\_APPEND\_EN |
| VI\_ATTR\_SEND\_END\_EN | VI\_ATTR\_GPIB\_PRIMARY\_ADDR |
| VI\_ATTR\_TERMCHAR | VI\_ATTR\_GPIB\_SECONDARY\_ADDR |
| VI\_ATTR\_TERMCHAR\_EN | VI\_ATTR\_GPIB\_REN\_STATE |
| VI\_ATTR\_TMO\_VALUE | VI\_ATTR\_GPIB\_ADDR\_STATE |
| VI\_ATTR\_DEV\_STATUS\_BYTE | VI\_ATTR\_CMDR\_LA |
| VI\_ATTR\_WR\_BUF\_OPER\_MODE | VI\_ATTR\_IO\_PROT |
| VI\_ATTR\_VXI\_LA | VI\_ATTR\_TRIG\_ID |

**SOCKET Resource**

|  |  |
| --- | --- |
| VI\_ATTR\_INTF\_NUM | VI\_ATTR\_WR\_BUF\_OPER\_MODE |
| VI\_ATTR\_INTF\_TYPE | VI\_ATTR\_DMA\_ALLOW\_EN |
| VI\_ATTR\_INTF\_INST\_NAME | VI\_ATTR\_RD\_BUF\_OPER\_MODE |
| VI\_ATTR\_SEND\_END\_EN | VI\_ATTR\_FILE\_APPEND\_EN |
| VI\_ATTR\_TERMCHAR | VI\_ATTR\_TCPIP\_ADDR |
| VI\_ATTR\_TERMCHAR\_EN | VI\_ATTR\_TCPIP\_HOSTNAME |
| VI\_ATTR\_TMO\_VALUE | VI\_ATTR\_TCPIP\_PROT |
| VI\_ATTR\_TCPIP\_NODELAY | VI\_ATTR\_IO\_PORT |
| VI\_ATTR\_TCPIP\_KEEPALIVE |  |
| VI\_ATTR\_RD\_BUF\_SIZE | VI\_ATTR\_WR\_BUF\_SIZE |

B.2 Summary of Events

**VISA Resource Template**

(These events are based on the VISA Resource Template and are available to all other resources.)

VI\_EVENT\_EXCEPTION

**INSTR Resource**

VI\_EVENT\_IO\_COMPLETION

VI\_EVENT\_SERVICE\_REQ

VI\_EVENT\_TRIG

VI\_EVENT\_VXI\_SIGP

VI\_EVENT\_VXI\_VME\_INTR

VI\_EVENT\_USB\_INTR

VI\_EVENT\_PXI\_INTR

**MEMACC Resource**

VI\_EVENT\_IO\_COMPLETION

**INTFC Resource**

VI\_EVENT\_GPIB\_CIC

VI\_EVENT\_GPIB\_TALK

VI\_EVENT\_GPIB\_LISTEN

VI\_EVENT\_CLEAR

VI\_EVENT\_TRIG

VI\_EVENT\_IO\_COMPLETION

VI\_EVENT\_SERVICE\_REQ

**BACKPLANE Resource**

VI\_EVENT\_TRIG

VI\_EVENT\_VXI\_VME\_SYSFAIL

VI\_EVENT\_VXI\_VME\_SYSRESET

**SERVANT Resource**

VI\_EVENT\_CLEAR

VI\_EVENT\_IO\_COMPLETION

VI\_EVENT\_GPIB\_TALK

VI\_EVENT\_GPIB\_LISTEN

VI\_EVENT\_TRIG

VI\_EVENT\_VXI\_VME\_SYSRESET

VI\_EVENT\_TCPIP\_CONNECT

**SOCKET Resource**

VI\_EVENT\_IO\_COMPLETION

B.3 Summary of Operations

**VISA Resource Template**

(These operations are based on the VISA Resource Template and are available to all other resources.)

viClose(vi)

viGetAttribute(vi,attribute,attrState)

viSetAttribute(vi,attribute,attrState)

viStatusDesc(vi,status,desc)

viTerminate(vi,degree,jobId)

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

viUnlock(vi)

viEnableEvent(vi,eventType,mechanism,context)

viDisableEvent(vi,eventType,mechanism)

viDiscardEvents(vi,eventType,mechanism)

viWaitOnEvent(vi,ineventType,timeout,outEventType,outContext)

viInstallHandler(vi,eventType,handler,userHandle)

viUninstallHandler(vi,eventType,handler,userHandle)

**VISA Resource Manager**

viOpenDefaultRM(sesn)

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

viFindRsrc(sesn,expr,findList,retcnt,instrDesc)

viFindNext(findList,instrDesc)

viParseRsrc(sesn, rsrcName, intfType, intfNum)

viParseRsrcEx(sesn, rsrcName, intfType, intfNum, rsrcClass, unaliasedExpandedRsrcName, aliasIfExists)

**INSTR Resource**

viRead(vi,buf,count,retCount)

viReadAsync(vi,buf,count,jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi,buf,count,retCount)

viWriteAsync(vi,buf,count,jobId)

viWriteFromFile(vi, fileName, count, retCount)

viAssertTrigger(vi,protocol)

viReadSTB(vi,status)

viClear(vi)

viSetBuf(vi,mask,size)

viFlush(vi,mask)

viPrintf(vi,writeFmt,arg1,arg2,...)

viVPrintf(vi,writeFmt,params)

viSPrintf(vi,buf,writeFmt,arg1,arg2,...)

viVSPrintf(vi,buf,writeFmt,params)

viBufWrite(vi,buf,count,retCount)

viScanf(vi,readFmt,arg1,arg2,...)

viVScanf(vi,readFmt,params)

viSScanf(vi,buf,readFmt,arg1,arg2,...)

viVSScanf(vi,buf,readFmt,params)

viBufRead(vi,buf,count,retCount)

viQueryf(vi,writeFmt,readFmt,arg1,arg2,...)

viVQueryf(vi,writeFmt,readFmt,params)

viIn8(vi,space,offset,val8)

viIn16(vi,space,offset,val16)

viIn32(vi,space,offset,val32)

viOut8(vi,space,offset,val8)

viOut16(vi,space,offset,val16)

viOut32(vi,space,offset,val32)

viMoveIn8(vi,space,offset,length,buf8)

viMoveIn16(vi,space,offset,length,buf16)

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

viMoveOut8(vi,space,offset,length,buf8)

viMoveOut16(vi,space,offset,length,buf16)

viMoveOut32(vi,space,offset,length,buf32)

viMove(vi,srcSpace,srcOffset,srcWidth,destSpace,destOffset,destWidth,length)

viMoveAsync(vi,srcSpace,srcOffset,srcWidth,destSpace,destOffset,destWidth,  
length,jobId)

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

viUnmapAddress(vi)

viPeek8(vi,addr,val8)

viPeek16(vi,addr,val16)

viPeek32(vi,addr,val32)

viPoke8(vi,addr,val8)

viPoke16(vi,addr,val16)

viPoke32(vi,addr,val32)

viMemAlloc(vi,size,offset)

viMemFree(vi,offset)

viGpibControlREN(vi,mode)

viVxiCommandQuery(vi,mode,cmd,response)

viUsbControlOut(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf)

viUsbControlIn(vi, bmRequestType, bRequest, wValue, wIndex, wLength, buf, retCnt)

**MEMACC Resource**

viIn8(vi,space,offset,val8)

viIn16(vi,space,offset,val16)

viIn32(vi,space,offset,val32)

viOut8(vi,space,offset,val8)

viOut16(vi,space,offset,val16)

viOut32(vi,space,offset,val32)

viMoveIn8(vi,space,offset,length,buf8)

viMoveIn16(vi,space,offset,length,buf16)

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

viMoveOut8(vi,space,offset,length,buf8)

viMoveOut16(vi,space,offset,length,buf16)

viMoveOut32(vi,space,offset,length,buf32)

viMove(vi,srcSpace,srcOffset,srcWidth,destSpace,destOffset,destWidth,length)

viMoveAsync(vi,srcSpace,srcOffset,srcWidth,destSpace,destOffset,destWidth,  
length,jobId)

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

viUnmapAddress(vi)

viPeek8(vi,addr,val8)

viPeek16(vi,addr,val16)

viPeek32(vi,addr,val32)

viPoke8(vi,addr,val8)

viPoke16(vi,addr,val16)

viPoke32(vi,addr,val32)

viMemAlloc(vi,size,offset)

viMemFree(vi,offset)

**INTFC Resources**

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, fileName, count, retCount)

viAssertTrigger(vi, protocol)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viBufRead(vi, buf, count, retCount)

viGpibControlREN(vi, mode)

viGpibControlATN (vi, mode)

viGpibPassControl(vi, primAddr, secAddr)

viGpibCommand(vi, buf, count, retCount)

viGpibSendIFC(vi)

**BACKPLANE Resources**

viAssertTrigger(vi, protocol)

viAssertUtilSignal(vi, line)

viAssertIntrSignal(vi, mode, statusID)

viMapTrigger(vi, trigSrc, trigDest, mode)

viUnmapTrigger(vi, trigSrc, trigDest)

viPxiReserveTriggers(vi, cnt, trigBuses, trigLines, failureIndex)

**SERVANT Resources**

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, fileName, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, fileName, count, retCount)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viBufRead(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params

viBufWrite(vi, buf, count, retCount)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)

viAssertIntrSignal(vi, mode, statusID)

viAssertUtilSignal(vi, line)

**SOCKET Resource**

viRead(vi, buf, count, retCount)

viReadAsync(vi, buf, count, jobId)

viReadToFile(vi, filename, count, retCount)

viWrite(vi, buf, count, retCount)

viWriteAsync(vi, buf, count, jobId)

viWriteFromFile(vi, filename, count, retCount)

viAssertTrigger(vi,protocol)

viReadSTB(vi,status)

viClear(vi)

viSetBuf(vi, mask, size)

viFlush(vi, mask)

viBufRead(vi, buf, count, retCount)

viScanf(vi, readFmt, arg1, arg2, ...)

viVScanf(vi, readFmt, params)

viPrintf(vi, writeFmt, arg1, arg2, ...)

viVPrintf(vi, writeFmt, params)

viBufWrite(vi, buf, count, retCount)

viSScanf(vi, buf, readFmt, arg1, arg2, ...)

viVSScanf(vi, buf, readFmt, params)

viSPrintf(vi, buf, writeFmt, arg1, arg2, ...)

viVSPrintf(vi, buf, writeFmt, params)