VPP-4.3.6:

VISA Implementation Specification for .NET

February 8, 2024
Revision 7.4

NOTICE

VPP-4.3.6: *VISA Implementation Specification for .NET* 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 IVI Foundation 3

Section 2: Overview of VISA.NET I/O Library Specification 1
2.1. Objectives of This Specification 2
2.2. Audience for This Specification 3
2.3. Scope and Organization of This Specification 4
2.4. Application of This Specification 5
2.5. References 6
2.6. Definition of Terms and Acronyms 7
2.7. Conventions 8

Section 3: VISA.NET Infrastructure 1
3.1. Target Operating Systems 2
3.2. Target Languages and Application Development Environments 3
3.3. Namespace Requirements 4
3.4. VISA.NET Data Types 5
3.4.1. Enumerations 5
3.4.2. Exceptions 5
3.5. VISA.NET Events and Asynchronous I/O 6
3.5.1. Hardware Events 6
3.5.2. Asyncronous I/O 6
3.6. VISA.NET Interfaces 7
3.7. Initializing a VISA.NET Session 8
3.7.1. The VISA.NET Global Resource Manager 8
3.7.2. Vendor Specific Resource Managers 8
3.7.3. Session Constructors 8
3.8. VISA.NET I/O Implementation and Distribution Requirements 9

Section 4: VISA.NET Data Types 1

Section 5: VISA.NET Enumerations 1
5.1. AccessMode 2
5.2. AddressSpace 3
5.3. AtnMode 4
5.4. BinaryEncoding 5
5.5. ByteOrder 6
5.6. DataWidth 7
5.7. EventQueueStatus 8
5.8. EventType 9
5.9. GpibAddressedState 10
5.10. GpibInstrumentRemoteLocalMode 11
5.11. GpibInterfaceRemoteLocalMode 12
5.12. HardwareInterfaceType 13
5.13. IOBuffers 14
5.14. IOProtocol 15
5.15. LineState 16
5.16. NativeVisaAttribute 17
5.17. PxiMemoryType 21
5.18. ReadStatus 22
5.19. RemoteLocalMode 23
5.20. ResourceLockState 24
5.21. ResourceOpenStatus 25
5.22. SerialFlowControlModes 26

5.23. SerialParity 27
5.24. SerialTerminationMethod 28
5.25. StatusByteFlags 29
5.26. SerialStopBitsMode 30
5.27. TriggerLine 31
5.28. TriggerLines 32
5.29. VxiAccessPriviledge 33
5.30. VxiCommandMode 34
5.31. VxiDeviceClass 35
5.32. VxiTriggerProtocol 36
5.33. VxiUtilitySignal 37

Section 6: VISA.NET Exceptions and Status Codes 1
6.1. Exception Overview 1
6.2. VISA.NET Exceptions 3
6.2.1. Ivi.Visa.VisaException 3
6.2.2. Ivi.Visa.IOTimeoutException 4
6.2.3. Ivi.Visa.NativeVisaException 5
6.2.4. Ivi.Visa.TypeFormatterException 6
6.3. NativeErrorCode Class 8
6.3.1. GetMacroNameFromStatusCode() 10

Section 7: VISA.NET Hardware Events 1
7.1. Hardware Event APIs 2
7.2. .NET Event Handlers 5
7.3. VISA.NET Event Arguments 6
7.3.1. VisaEventArgs Class 7
7.3.2. GpibControllerInChargeEventArgs 9
7.3.3. PxiInterruptEventArgs 10
7.3.4. UsbInterruptEventArgs 11
7.3.5. VxiSignalProcessorEventArgs 12
7.3.6. VxiTriggerEventArgs 13
7.3.7. VxiInterruptEventArgs 14
7.3.8. INativeVisaEventArgs Interface 15
7.4. Vendor Defined Events 17
7.5. Event Methods 18

Section 8: VISA.NET Sessions 1
8.1. Session Overview 1
8.1.1. Resources and Resource Descriptors 1
8.1.2. Resources Managers 1
8.1.3. Session Interfaces 1
8.1.4. Locking 2
8.2. Session Interfaces 3
8.3. IVisaSession Interface 4
8.3.2. SynchronizeCallbacks 8
8.4. INativeVisaSession Interface 9

Section 9: Message Based Session Interfaces 1
9.1. IMessageBasedSession Interface 1
9.2. IMessageBasedRawIO 3
9.2.1. Synchronous I/O 4
9.2.2. Asynchronous I/O 9
9.3. Custom Formatting 22
9.3.1. Type Formatters 22
9.3.2. ITypeFormatter Interface 24

9.4. IMessageBasedFormattedIO 28
9.4.2. BinaryEncoding 30
9.4.3. ReadBufferSize 31
9.4.4. WriteBufferSize 32
9.4.5. TypeFormatter 33
9.4.6. DiscardBuffers 34
9.4.7. FlushWrite 35
9.4.8. Printf Format Strings 36
9.4.9. Printf 46
9.4.10. PrintfAndFlush 47
9.4.11. PrintfArray 48
9.4.12. PrintfArrayAndFlush 49
9.4.13. Scanf Format Strings 50
9.4.14. Scanf 59
9.4.15. ScanfArray 62
9.4.16. Introduction to Formatted Write Methods 63
9.4.17. Write 64
9.4.18. WriteLine 65
9.4.19. WriteList 66
9.4.20. WriteLineList 68
9.4.21. WriteBinary 70
9.4.22. WriteBinary AndFlush 73
9.4.23. Introduction to Formatted Read Methods 76
9.4.24. ReadString 77
9.4.25. Read 78
9.4.26. ReadLine (String) 79
9.4.27. ReadLine 80
9.4.28. ReadList 81
9.4.29. ReadLineList 83
9.4.30. ReadBinaryBlock 85
9.4.31. ReadLineBinaryBlock 89
9.4.32. ReadWhileMatch 93
9.4.33. ReadUntilMatch 94
9.4.34. ReadUntilEnd 95
9.4.35. Introduction to Formatted Skip Methods 96
9.4.36. Skip 97
9.4.37. SkipString 98
9.4.38. SkipUntilEnd 99
9.5. FormattedIO Implementations 100
9.5.2. MessageBasedFormattedIO Constructors 101

Section 10: Register Based Session Interfaces 1
10.1. IRegisterBasedSession 2
10.2. IMemoryMap 5

Section 11: INSTR Resources 1
11.1. IGpibSession 1
11.2. IPxiSession 3
11.3. ISerialSession 6
11.4. ITcpipSession 8
11.5. IUsbSession 10
11.6. IVxiSession 12

Section 12: MEMACC Resources 1
12.1. IPxiMemorySession 2
12.2. IVxiMemorySession Interface 3

Section 13: INTFC Resources 1
13.1. IGpibInterfaceSession Interface 1

Section 14: SOCKET Resources 1
14.1. ITcpipSocketSession 1

Section 15: BACKPLANE Resources 1
15.1. IPxiBackplaneSession 2
15.2. IVxiBackplaneSession 4

Section 16: VISA.NET I/O Conflict Resolution 1

Section 17: Resource Manager Classes 1
17.1. The Vendor-Specific Resource Manager Component 2
17.2. IResourceManager Interface 4
17.3. The Global Resource Manager (GRM) Component 5
17.4. GlobalResourceManager Class 6
17.5. ParseResult Class 8

Section 18: VISA.NET Installation 1
18.1. VISA.NET Shared Components 1
18.2. Vendor-Specific VISA.NET Installer Requirements 2
18.2.1. Prerequisites 2
18.2.2. VISA.NET Implementation Location 2
18.2.3. VISA.NET Registry Entries 2
18.3. VISA.NET Resource Manager Registration 3
18.3.2. General Installation Requirements for Vendor Specific Components 3

Section 19: Version Control 1
19.1. VISA.NET Shared Components 1
19.1.1. Versioning with Policy Files 1
19.1.2. Maintaining Software Configurations 2
19.1.3. Versioning for Policy Files 2
19.1.4. Naming New Versions of .NET Types 2
19.1.5. Versioning Enumerations 3
19.1.6. Versioning Interfaces 3
19.1.7. Versioning Classes 4
19.1.8. Other Considerations 5
19.2. VISA.NET Shared Components Installer 5
19.3. VISA.NET Implementations 5

IVI VISA.NET Revision History

This section is an overview of the revision history of the IVI VISA.NET specification.

Table 1. IVI VISA.NET Class Specification Revisions

Status	Action
Revision 5.4 June 19, 2014	First version of specification.
Revision 5.5 February 11, 2015	A variety of editorial and minor changes to clarify details and synchronize with the VISA.NET Shared Components.
Revision 5.5 August 6, 2015	Removed Windows 2000 and added Windows 10 to the list of supported operating systems.
Revision 5.7 February 26, 2016	Added PXI Trigger lines TTL8-TTL11. A variety of editorial changes to clarify details and synchronize with the VISA.NET Shared Components.
Revision 5.8 June 7, 2016	Removed Windows XP and Windows Vista from the list of supported operating systems.
Revision 7.2 May 19, 2022	Add support for HiSLIP 2.0. Add support for secure networked connections.
Revision 7.3 December 19, 2022	Added Windows 11 to the list of supported operating systems.
Revision 7.4 October 30, 2023 February 8, 2024	Added support for .NET (6+) versions of .NET. Post-review editorial changes: Change all preprocessor uses ofNET5_0_OR_GREATER to NET6_0_OR_GREATER Remove the async keyword from interfaces.

When a specification in the following list is revised, the version must be identical to the version of any other specifications in the list that are revised at the same time. (This accounts for the initial specification version of this specification.)

- VPP-4.3
- VPP-4.3.2
- VPP-4.3.3
- VPP-4.3.4
- VPP-4.3.5
- VPP-4.3.6

Section 1: Introduction to the IVI Foundation

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. The primary purpose of the Consortium is to promote the development and adoption of standard specifications for programming test instrument capabilities; to focus on the needs of the people that use and develop test systems who must take off-the-shelf instrument drivers and build and maintain high-performance test systems; to build on existing industry standards to deliver specifications that simplify interchanging instruments and provide for better performing and more easily maintainable programs that use IVI drivers.

The VISA Implementation Specification for .NET (VPP-4.3.6) 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

Trademarks

Section 2: Overview of VISA.NET I/O Library Specification

This section introduces the VISA.NET specification. The VISA.NET specification is a document authored by the IVI Foundation. The technical work embodied in this document and the writing of this document was performed by the VISA.NET Technical Working Group.

This section provides a complete overview of the VISA.NET I/O 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.NET I/O specification describes the VISA.NET I/O architectural model, the configuration model, the VISA.NET interface definitions, and their semantics. In cases where the semantics mirror functionality in VISA, there will be an annotated link to VPP4-3, *The VISA Library Specification*. In cases where VISA.NET supplies new functionality, the semantics will be described in this specification.

2.2. Audience for This Specification

The primary audience is I/O vendors who wish to implement and install VISA-compliant I/O software.

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: *Introduction to the IVI Foundation*, describes the IVI Foundation.

Section 2: *Overview of VISA.NET I/O Library Specification*, provides an overview of this specification, including the objectives, scope and organization, application, references, definition of terms and acronyms, and conventions.

Section 3: *VISA.NET Infrastructure*, describes aspects of the VISA.NET API and implementations that distinguish them from either VISA C or VISA COM.

Section 4: *VISA.NET Data Types*, describes the data types that may be used in VISA.NET.

Section 5: *VISA.NET Enumerations*, describes the enumerations that are defined by VISA.NET.

Section 6: *VISA.NET Exceptions and Status Codes*, explains how VISA.NET uses exceptions, describes the exceptions that are defined by VISA.NET, including the status codes that may be used with the Native VISA exception.

Section 7: *VISA.NET Hardware Events*, describes the events that are defined in VISA.NET to report various hardware-related events.

Section 8: *VISA.NET Sessions*, presents an overview of VISA.NET sessions, and describes the base VISA session interface.

Section 9: *Message Based Session Interfaces*, describes the base message-based session interfaces used for message-based protocols, and in particular the interfaces used for both raw (unformatted) and formatted I/O.

Section 10: *Register Based Session Interfaces*, describes the base register-based session interfaces used for register-based protocols.

Section 11: *INSTR Resources*, describes the session interfaces used for INSTR resources.

Section 12: *MEMACC Resources*, describes the session interfaces used for MEMACC resources.

Section 13: *INTFC Resources*, describes the session interfaces used for INTFC resources.

Section 14: *SOCKET Resources*, describes the session interfaces used for SOCKET resources.

Section 15: *BACKPLANE Resources*, describes the session interfaces used for BACKPLANE resources.

Section 16: *VISA.NET I/O Conflict Resolution*, references the conflict resolution process used for selecting a particular VISA.NET implementation for a particular resource in cases where implementations from multiple vendors are available.

Section 17: *Resource Manager Classes*, describes the Global Resource Manager and vendor-specific resource managers.

Section 18: *VISA.NET Installation*, describes installation details for both VISA.NET Shared Components and VISA.NET implementations.

Section 19: *Version Control*, describes how VISA.NET Shared Components and VISA.NET implementations are versioned.

2.4. Application of This Specification

This specification is intended for use by developers of VISA.NET I/O Libraries software, by developers of instrument driver that use VISA.NET to communicate with instruments, and by developers who wish to use VISA.NET directly in their programs.

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*
- *NI-488.2 User Manual for DOS*, National Instruments Corporation
- *NI-488.2M User Manual*, National Instruments Corporation
- *NI-VXI Programmer Reference Manual*, National Instruments Corporation
- *NI-VXI User Manual*, National Instruments Corporation
- *ANSI/IEEE Standard 1174-2000, Standard Serial Interface for Programmable Instrumentation*
- IVI-6.1, *IVI High-Speed LAN Instrument Protocol (HiSLIP)*, Revision 1.1, IVI Foundation
- VPP-2, *System Frameworks Specification*
- VPP-4.3, *The VISA Library*
- VPP-4.3.2, *VISA Implementation Specification for Textual Languages*
- VPP-4.3.3, *VISA Implementation Specification for the G Language*
- VPP-4.3.4, *VISA Implementation Specification for COM*
- VPP-4.3.5, *VISA Shared Components*
- VPP-6, *Installation and Packaging Specification*
- VPP-9, *Instrument Vendor Abbreviations*
- VXI-1, *VXIbus System Specification*, Revision 1.4, VXIbus Consortium
- VXI-11, *TCP/IP Instrument Protocol*, VXIbus Consortium

2.6. Definition of Terms and Acronyms

The following are some commonly used terms within this document. This section does not include terms that are defined in VPP-4.3, *The VISA Library*. Please refer to that document for a list of generally applicable VISA terms.

.NET **A** Microsoft technology for reusable software components.

.NET Class A software construct defined by Microsoft's .NET specification that represents a logical object and derives from System.Object. Note that classes are reference types.

.NET (6+) .NET (6+) refers to .NET versions 6 and after. In this specification, we will only use ".NET (6+)" when using ".NET" must be differentiated as referring to .NET (6+) and not .NET Framework.

.NET Delegate A special .NET type that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback.

.NET Event A message sent by an object to signal the occurrence of an action. The object that raises the event is the event sender. The object that captures the event and responds to it is the event receiver.

.NET Exception .NET operations indicate failure by throwing exceptions.
The runtime implements language independent exceptions that may be thrown across process and even machine boundaries. The techniques for catching exceptions are specific to each language.

.NET Framework In this specification, se ".NET Framework" is only used to refer to syntax and behavior that applies only to .NET Framework.

.NET Interface A specification of a group of related features (events, methods, properties, and so on) containing additional marshalling and other information, but with no implementation in C#. .NET Classes may implement one or more interfaces, in which case they must implement all of the features defined by the interfaces.

.NET Object A live instance of a .NET Class.

.NET Property A "smart field" with a private data member accompanied by accessor functions, which is accessed syntactically as a field of a class. Note that .NET properties are used in VISA.NET to implement VISA attributes.

Application Policy File A policy file that specifies policies that are applied to a specific application. For example, an application policy file may be used to redirect the applications references from an earlier version of a referenced assembly to a later version. Application policy files have a higher priority than publisher policy files, but lower priority than machine policy files.

Assembly A DLL or EXE that includes .NET executable code. A VISA.NET vendor-specific I/O Assembly is always a DLL (and additionally requires that at least one instantiatable class implement the interface "`IVisaSession`").

Attribute A value within a resource that reflects a characteristic of the operational state of a resource. Also known as a property.

Machine Policy File A policy file that specifies policies that are applied to all application or components on a particular PC. For example, a machine policy file may be used to redirect all references on a particular PC from an earlier version of a referenced assembly to a later version. Machine policy files have the highest priority.

Publisher Policy File	A policy file that specifies policies that the publisher intends to be applied to a published assembly. For example, a publisher policy file may be used to redirect all references from an earlier version of an assembly to a later version. Publisher policy files have the lowest priority.
Side-by-Side Installation	The ability to install two different versions of the same assembly at the same time on a single PC.

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 implementers 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 implementer 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.NET Infrastructure

The VISA.NET I/O API has a few rules that are unique to VISA.NET, that apply across all the interfaces and components. Most of these rules reflect the ways in which .NET technology differs fundamentally from ANSI C or Microsoft COM, or standard .NET patterns for writing .NET code. Some of the rules reflect a conscious choice by the IVI Foundation to support particular .NET alternatives where .NET itself or the standard patterns are ambiguous. This specification documents these differences.

VISA.NET does not support some of the features of VISA.

- SERVANT resources are not supported in VISA.NET because it is so rarely used and does not lend itself to the kind of vendor-independent interoperability for which VISA was designed.
- File based I/O methods are not supported in VISA.NET because the .NET framework defines very capable file I/O classes that are easy to use with VISA.NET's other I/O methods.

3.1. Target Operating Systems and Frameworks

VISA.NET (6+)

VISA.NET (6+) implementations work on one or more of the following Microsoft operating systems: Windows 10 (64-bit editions), and Windows 11. IVI supports Windows-on-Windows 32-bit apps on 64-bit operating systems.

VISA.NET Framework

VISA.NET Framework implementations work on one or more of the following Microsoft operating systems: Windows 8 (32 and 64-bit editions), Windows 10 (32 and 64-bit editions), and Windows 11.

VISA.NET implementations may also work on other versions of Windows, as qualified by VISA.NET vendors.

For the minimum service pack level required to use the VISA.NET Shared Components on each operating system, refer to the download page on the IVI Foundation web site, www.ivifoundation.org.

3.1.1. .NET Versions and Frameworks

With the introduction of .NET 5, Microsoft made some significant changes to .NET to better support multi-platform adoption. Older versions of .NET (2.0 through 4.x) are referred to as.NET Framework, while versions 5.0 and greater are simply known as .NET or, if additional clarity is needed, ".NET (6+)".

Versions of the VISA.NET shared components through version 7.2 are .NET Framework versions that target .NET 2.0.

The VISA.NET shared components version 7.3 and later are available in both .NET Framework versions that target .NET 4.5, and .NET versions that target .NET 6.0 or after.

There are differences between the VISA.NET Framework API and the newer VISA.NET API. In general, this specification uses ".NET" and ".NET Framework" as follows.

- *.NET* is used to refer to all content related to the .NET API, including content that is identical to the .NET Framework, as well as content that is unique to .NET (6+).
- *.NET (6+)* is used where it is important to emphasize something that does not apply to .NET Framework.
- *.NET Framework* is used when identifying content that is unique to the .NET Framework API, to distinguish it from the ".NET" API.

3.1.2. Preprocessor Directives

.NET C# code uses preprocessor directives to indicate code that is unique to either .NET or .NET Framework. The directives are used in this specification to clarify code specific to a version.

Code that is only used for .NET Framework APIs uses the directive `#if NETFRAMEWORK`.

Code that is not used for .NET Framework APIs uses the directive `#if NET6_0_OR_GREATER`.

Code that is not bracketed by one of these `#if` directives is used for .NET in general, including .NET Framework.

3.2. Target Languages and Application Development Environments

VISA.NET works in the target languages and application development environments listed in Table 4-3.

Table-3.1 Target languages and ADEs for VISA.NET

32-bit	Native 64-bit
Agilent VEE	
MathWorks MATLAB	MathWorks MATLAB
Microsoft Visual Basic .NET	Microsoft Visual Basic .NET
Microsoft Visual C#	Microsoft Visual C#
Microsoft Visual C++	Microsoft Visual C++
National Instruments LabVIEW	National Instruments LabVIEW
National Instruments LabWindows/CVI	National Instruments LabWindows/CVI

VISA.NET complies with the Common Language Specification (CLS), with the exception of the use of unsigned integer types. In principle, VISA.NET can work in other development environments in which the .NET CLR and unsigned integers are supported.

3.3. Namespace Requirements

The primary IVI VISA.NET namespace is `Ivi.Visa`. All of the VISA.NET managed code documented in this specification and in VPP-4.3.5, *VISA Shared Components*, is defined in the `Ivi.Visa` namespace.

The Ivi.Visa assembly does contain other undocumented, public namespaces. The code in these undocumented namespaces contains internal implementation details and is not intended for public use. The IVI Foundation does not take any responsibility to support or document any of the source code contained in these undocumented namespaces. Source is available to IVI member companies. IVI member companies may use this code in their VISA.NET implementations.

IMPLEMENTATION

RULE 3.3.1

All VISA.NET implementations **SHALL** use the namespace <*vendor*>.*Visa*, where vendor is the name of the vendor. For example, `NationalInstruments.Visa`.

3.4. VISA.NET Data Types

VISA.NET includes most of the VISA data types, but it also includes a large number of data types that do not exist in VISA. VISA.NET takes advantage of the strongly typed nature of .NET to provide data types for enumerations, exceptions, event handlers and event arguments, interfaces that define standard functionality, and even some classes.

3.4.1. Enumerations

Enumerations specify a limited set of named constants.

In VISA C enumerations are represented by constants with similar names, each of which is assigned a particular value, but used with integer variables. The integer variables were not limited to the values of the defined constants, but can be assigned any value.

In VISA.NET, enumerations are used with variables that are strongly typed to the variable, so that values that are not defined as part of the enumeration cannot be assigned to the variable.

IMPLEMENTATION

RULE 3.4.1

All VISA.NET enumerations, including vendor defined enumerations, **SHALL** have a member whose value is zero, unless the enumeration explicitly maps to values in the VISA C specification.

3.4.2. Exceptions

.NET exceptions provide an elegant way of reporting errors to calling programs. Exceptions include more information about errors, and require less processing until the calling code is actually ready to deal with them.

Unlike VISA C and VISA COM, which use method return codes or HRESULTS to return a VISA status code, VISA.NET uses exceptions rather than return codes to report errors to callers. Exceptions generally include a variety of information about the error that occurred and propagate up the call stack automatically until they are handled. With exceptions there is no need to check a return code after every call to see if an error occurred.

While the VISA specification made an attempt to describe all of the status codes that could be returned by a function, this is impractical for the VISA.NET specification for several reasons. The primary reason is that implementation of VISA.NET features can vary enough that trying to catalog errors for interfaces is not possible. Another reason is that some standard exceptions are so common that documenting them for every method would clutter the documentation. Documentation of exceptions is better left to the implementation's documentation than the specification for these reasons.

VISA.NET uses .NET Framework exceptions extensively. All exceptions (including all .NET Framework exceptions) derive from `System.Exception`, which provides a common set of capabilities to all exceptions. Derived exceptions may only differ from this base exception by a name that indicates the reason for the exception. Other exceptions add additional properties for information

VISA.NET defines several exceptions, all of which derive from `Ivi.Visa.VisaException`n. This makes it possible for client code to catch all VISA.NET specific exceptions by testing for a single exception type (`Ivi.Visa.VisaException`).

VISA C and VISA COM may also return positive return codes to indicate a success condition or warning. VISA.NET does not use exceptions to return this type of information. Where needed, this information is returned via output arguments to specific methods.

RULE 3.4.2

All VISA.NET error conditions **SHALL** be communicated to the calling program using exceptions.

3.5. VISA.NET Events and Asynchronous I/O

VISA.NET includes mechanisms for hardware events such as interrupts and trigger notifications. In addition there is a set of mechanisms to support asynchronous I/O notification that provides a flexible set of ways to notify users that asynchronous I/O has completed.

3.5.1. Hardware Events

VISA C provides a number of functions that enable calling programs to register for and receive notification of hardware events. Each of these functions has an event type parameter that identifies the event.

VISA.NET provides equivalent functionality in two forms. First, it provides a similar set of methods that allow for blocking waits when asynchronous events are being used. Second, it provides a number of .NET events, which do not support blocking waits.

3.5.1.1. .NET Events

.NET events provide callback delegates, registration methods, and a notification mechanism that are specific to particular events. Event handlers are declared in interfaces. Note that .NET events do not implement a blocking wait mechanism, but are recommended when blocking waits are not required.

In VISA.NET, asynchronous events map to .NET events. When using .NET events, registering for the event corresponds to calling `viInstallHandler` and `viEnableEvent` with the event mechanism of `VI_HNDLR` and unregistering the event corresponds to calling `viDisableEvent` and `viUninstallHandler`. There are no equivalents to `viDiscardEvent` and `viWaitOnEvent` when using .NET events. Events are received by registered clients when the event is fired, and can be ignored if needed.

3.5.1.2. Event Methods

All types of VISA.NET sessions contain methods that allow the calling program to enable and disable events, discard event notifications that are not needed, and most importantly, wait on an event. Each method includes an `eventType` argument that indicates the kind of event to which the method applies. Event methods are recommended when blocking waits are required.

The VISA.NET methods are similar to the ones found in VISA C. The VISA.NET version of `EnableEvent` corresponds to `viEnableEvent` called with the event mechanism of `VI_QUEUE`. The `DisableEvent`, `Discard Event`, and `WaitOnEvent` methods correspond exactly to the corresponding VISA C functions.

3.5.2. Asyncronous I/O

VISA.NET Raw I/O includes methods for asynchronous operations. By using these methods, calling programs can do other tasks while waiting for I/O to complete, and multiple I/O operations can be queued on different sessions.

There are three ways that a calling program can determine when an asynchronous I/O operation is complete - polling, blocking waits, and callbacks. Note that there is not a .NET event for Asynchronous I/O completion.

3.6. VISA.NET Interfaces

A session in VISA.NET creates and manages a communication channel to I/O hardware, or to an instrument (or other device) via I/O hardware. Sessions are specific to the type of connection being used (for example, GPIB, PXI, or USB). Sessions can also be specific to a connection to an attached device via an I/O Protocol (for example, a PXI session) or something that manages aspects of the connection (for example, a PXI backplane).

The VISA.NET standard defines *session interfaces* that define the APIs for VISA.NET's I/O sessions.

All session interfaces include some elements that are common to all sessions. These elements are defined in the interface `IVisaSession`. All VISA.NET session interfaces ultimately derive from `IVisaSession`, and so include this common functionality.

In general, session interfaces are either message-based or register-based. All message-based session interfaces include some elements common to all message-based sessions. These elements are defined in the interface `IMessageBasedSession`. All VISA.NET message-based session interfaces derive from `IMessageBasedSession`, and so include this common functionality.

Likewise, all register-based session interfaces include some elements common to all register-based sessions. These elements are defined in the interface `IRegisterBasedSession`. All VISA.NET register-based session interfaces derive from `IRegisterBasedSession`, and so include this common functionality.

3.7. Initializing a VISA.NET Session

VISA.NET sessions may be initialized in two ways. First, a resource manager may be used to initialize a session. Second, a vendor specific session may be initialized directly using a constructor provided for the class.

3.7.1. The VISA.NET Global Resource Manager

The VISA.NET Global Resource Manager (GRM) is part of the VISA.NET Shared Components. The VISA.NET GRM initializes a session that is based on a client-supplied resource name and is capable of connecting to the resource from among available implementations.

Where multiple implementations (particularly multiple vendor's implementations) are available, the GRM selects based on several criteria. These criteria may be determined by users, but there are also generally suitable defaults provided by the GRM.

Refer to Section 17.3, *The Global Resource Manager (GRM) Component* for details on the VISA.NET Resource Manager.

3.7.2. Vendor Specific Resource Managers

Each vendor must supply a vendor specific resource manager as part of their VISA.NET implementation. Vendor specific resource managers initialize a session based on a client-supplied resource name, but are only capable of connecting to resources that are supported by their implementation of VISA.NET.

Refer to Section 17.1, *The Vendor-Specific Resource Manager Component* for details on the VISA.NET Resource Manager.

3.7.3. Session Constructors

Implementations of various session interfaces are, by definition, vendor specific, including any constructors provided to initialize the session class. Vendors may provide several ways of initializing session classes and make appropriate recommendations to users.

3.8. VISA.NET I/O Implementation and Distribution Requirements

VISA.NET I/O Implementations will redistribute several shared global files and will also provide some vendor-specific components. The very minimum compliant installation would include the VISA.NET Shared Components, and provide a Vendor-Specific Resource Manager (SRM) with one VISA.NET I/O Resource Component that implements IVisaSession.

Example 1:

If a vendor wanted to provide a driver for a PC plug-in card that allowed SCPI string communication, it would redistribute the VISA.NET Shared Components, provide a resource manager that knows how to instantiate the plug-in's session class, and provide a VISA.NET I/O session class for the plug-in that implements IVisaSession and IMessageBasedSession interfaces.

Example 2:

If a vendor wished to provide a VISA.NET I/O implementation that could create ASRL INSTR and GPIB INSTR sessions, they would redistribute the VISA.NET Shared Components and provide an SRM that can parse both kinds of address strings and can find and create resources of both types. They would also provide two different VISA.NET I/O Resource Components, one that implemented ISerial, the IMessage interfaces, and the two base interfaces and another that implemented IGpib, the IMessage interfaces, and the two base interfaces.

The installation rules and requirements for the VISA.NET Shared Components are listed in *VPP-4.3.5 VISA Shared Components*.

In addition to the shared global files, a VISA.NET I/O implementation must provide several vendor-specific files to be compatible with the VISA.NET I/O standard.

Table 2.6.1 shows a list of the required files and some optional files.

Component Name	Description
Vendor-Specific Resource Manager (SRM)	An assembly containing a resource manager that can find and instantiate all of the resources implemented by the vendor's VISA.NET I/O implementation.
One or more Resource Components	One or more assemblies containing one or more classes that implement at least the IVisaSession interface.
Vendor Help File (optional)	A help file containing entries describing the errors returned by the Vendor's resources, information about the resources themselves, descriptions of any vendor-defined classes, and any additional information deemed appropriate by the vendor.

Table 2.6.1

The installation rules and requirements for the Vendor Specific Components are listed in Section 18.2, *Vendor-Specific VISA.NET Installer Requirements*

OBSERVATION 3.8.1

Unlike VPP-4.3.2 and VPP-4.3.3, which rely on a single file named visa32.dll, a VISA.NET I/O implementation has no name requirements. This allows both .NET-based and non-.NET-based implementations to reside side-by-side on the same system.

RECOMMENDATION 3.8.2

If a vendor provides both a VISA C and a VISA.NET implementation, and the VISA.NET implementation invokes a VISA C implementation, the recommendation is that the VISA.NET implementation invokes that vendor's VISA C implementation.

PERMISSION 3.8.1

If a vendor provides both a VISA C and a VISA.NET implementation, and the VISA.NET implementation invokes a VISA C implementation, the vendor's VISA.NET implementation may invoke any suitable VISA C implementation. This permission is necessary because the resource manager may need to select VISA-C DLLs based on resource type, and that happens after the VISA.NET assembly has been loaded.

OBSERVATION 3.8.2

From a user's perspective, VISA "operations" in their program may follow a "hybrid" path. If a vendor provides both VISA.NET and VISA-C, that vendor's VISA.NET may invoke their own VISA-C or may take the hybrid approach.

Section 4: VISA.NET Data Types

VISA defines a relatively limited set of data types compared to VISA.NET, but since it is important to understand the relationship between VISA and VISA.NET, it is important to understand how the VISA types relate to VISA.NET types.

Some basic types including Booleans, numbers, characters, and strings map more or less directly to their corresponding .NET types. Note that pointer types in VISA map to the corresponding scalar types in VISA.NET.

VISA.NET defines a variety of new types, including, but not limited to, enumerations, exceptions, events and event arguments, and interfaces. The types that are new to VISA.NET are defined in the following sections.

The following table identifies VISA.NET types that correspond to the VISA types defined in VPP-4.3.

VISA.NET Data Type	Description	Corresponding VISA Types
`System.Object`	The base class for all IVI.NET classes. Find Lists are returned by viFindResource and used by viFindNext – will have wait to see what the VISA.NET API looks like.	`ViObject, ViPObject`
`System.Object[]`	Variable Arguments – an array of objects	`ViVAList`

`System.Byte`	An 8-bit unsigned integer.	`ViUInt8, ViPUInt8`
		`ViInt8, ViPInt8`
		`ViByte, ViPByte`

`System.Byte[]`	An array of 8-bit unsigned integers.	`ViAUInt8, ViAInt8`
		`ViAByte`
		`ViBuf, ViPBuf`

`System.Int16`	A 16-bit signed integer. VISA.NET may use signed integers where VISA used unsigned.	`ViInt16, ViPInt16` `ViUInt16, ViPUInt16`
`System.UInt16`	A 16-bit unsigned integer.	`ViUInt16, ViPUInt16`
`System.Int16[]`	An array of 16-bit signed integers.	`ViAInt16, ViAUInt16`
`System.Int32`	A 32-bit signed integer. VISA.NET may use signed integers where VISA used unsigned.	`ViInt32, ViPInt32` `ViUInt32, ViPUInt32`
`System.UInt32`	A 32-bit unsigned integer.	`ViUInt32, ViPUInt32`
`System.Int32[]`	An array of 32-bit signed integers.	`ViAUInt32, ViAInt32`

`System.Int64`	A 64-bit signed integer.	`ViInt64, ViPInt64`
		`ViUInt64, ViPUInt64`
		`ViBusAddress,`
		`ViPBusAddress`
		`ViBusAddress64,`
		`ViPBusAddress64`
		`ViBusSize`

`System.Int64[]`	An array of 64-bit signed integers.	`ViAInt64,ViAUInt64`
`System.Single`	A 32-bit single-precision value.	`ViReal32, ViPReal32`
`System.Single[]`	An array of 32-bit single-precision values.	`ViAReal32`
`System.Double`	A 64-bit double-precision value.	`ViReal64, ViPReal64`
`System.Double[]`	An array of 64-bit double-precision values.	`ViAReal64`
`System.Boolean`	A type with two possible values: true and false.	`ViBoolean, ViPBoolean`

`System.Boolean[]`	An array of Boolean.	`ViABoolean`
`System.Char`	A Unicode character	`ViChar, ViPChar`

`System.String`	A Unicode string.	`ViString, ViPString`
		`ViConstString`
		`ViAChar`
		`ViRsrc, ViPRsrc`
		`ViKeyId, ViPKeyId`

`System.String[]`	An array of Unicode strings.	`ViAString, ViARsrc,`
		`ViFindList,`
		`ViPFindList`

`System.Version`	Resource version information.	`ViVersion, ViPVersion`
`.NET delegate`	A value representing an entry point to a VISA C operation for use as a callback when using C interop.	`ViHndlr`
`Ivi.Visa.AccessMode`	An enumeration of the different mechanisms that control access to a resource. Refer to Section 5.1, *AccessMode* for the definition.	`ViAccessMode,` `ViPAccessMode`
`Ivi.Visa.` `IVisaAsyncResult`	A reference to the result of an asynchronous I/O operation.	`ViJobId, ViPJobId`
`Ivi.Visa.EventType`	An enumeration of the possible types for an event. Refer to Section 5.8, *EventType* for the definition.	`ViEventType,` `ViPEventType`
`Ivi.Visa.EventType[]`	An array of Event Types.	`ViAEventType`
`Ivi.Visa.` `NativeErrorCode`	A class that contains the standard error status codes.	`ViStatus, ViPStatus`
`IMemoryMap`	In VISA.NET, address pointers are replaced by object references.	`ViAddr, ViPAddr`
A reference to a session class or to `Ivi.Visa.` `IResourceManager`.	In VISA.NET, sessions are replaced by instances of classes.	`ViSession, ViPSession`
`Ivi.Visa.` `NativeVisaAttribute`	In VISA.NET, attributes are implemented as properties. Refer to Section 5.16, *NativeVisaAttribute* for more information.	`ViAttr, ViPAttr`
`N/A`	This is the type that you are setting the attribute to, or that you expect to get. In VISA.NET, this is accomplished by the Set/Get overloads for the supported data types.	`ViAttrState,` `ViPAttrState`

`Ivi.Visa.`
`VisaEventArgs`
or `Ivi.Visa.`
`NativeEventArgs`
(or a derived class)
	In VISA.NET, specific events returned from WaitOnEvent are identified by the instance of VisaEventArgs returned. This is also true of handlers that are called with an instance of the event args.	`ViEvent, ViPEvent`

Section 5: VISA.NET Enumerations

VISA.NET defines the following enumerations. All enumerations are defined in the Ivi.Visa namespace.

- `AccessModes`
- `AddressSpace`
- `AtnMode`
- `BinaryEncoding`
- `ByteOrder`
- `DataWidth`
- `EventQueueStatus`
- `EventType`
- `GpibAddressedState`
- `GpibInstrRemoteLocalMode`
- `GpibInterfaceRemoteLocalMode`
- `HardwareInterfaceType`
- `IOBuffers`
- `IOProtocol`
- `LineState`
- `NativeVisaAttribute`
- `PxiMemoryType`
- `ReadStatus`
- `RemoteLocalMode`
- `ResourceLockState`
- `ResourceOpenStatus`
- `SerialFlowControlModes`
- `SerialParity`
- `SerialTerminationMethod`
- `StatusByteFlags`
- `StopBitMode`
- `TriggerLine`
- `TriggerLines`
- `VxiAccessPrivilege`
- `VxiCommandMode`
- `VxiDeviceClass`
- `VxiTriggerProtocol`
- `VxiUtilitySignal`

5.1. AccessMode

DEFINITION

```
[Flags]
public enum AccessMode
{
   None = 0,
   ExclusiveLock = 1,
   LoadConfig = 2
}
```

OBSERVATION 5.1.1

The `AccessMode` enumeration indicates the modes by which the resource specified in the Open method is to be accessed. Multiple access modes may be specified by combining multiple values. This enumeration corresponds to the defined values for the `accessMode` parameter in VISA's `viOpen` functions.

5.2. AddressSpace

DEFINITION

```
public enum AddressSpace
{
   VxiA16 = 0,
   VxiA24 = 1,
   VxiA32 = 2,
   VxiA64 = 3,
   PxiConfiguration = 4,
   PxiBar0 = 5,
   PxiBar1 = 6,
   PxiBar2 = 7,
   PxiBar3 = 8,
   PxiBar4 = 9,
   PxiBar5 = 10,
   PxiAllocation = 11
}
```

OBSERVATION 5.2.1

The `AddressSpace` enumeration indicates the bus address space used by VXI or PXI devices. This enumeration corresponds to the defined values for the `space` parameter that all register based operation include.

5.3. AtnMode

DEFINITION

```
public enum AtnMode
{
   Deassert = 0,
   Assert = 1,
   DeassertHandshake = 2,
   AssertImmediate = 3
}
```

OBSERVATION 5.3.1

The `AtnMode` enumeration indicates how to modify the state of the GPIB ATN (ATtentioN) interface line. This enumeration corresponds to the defined values for the parameter `mode` in the VISA function `viGpibControlATN`.

5.4. BinaryEncoding

DEFINITION

```
enum BinaryEncoding
{
   DefiniteLengthBlockData = 0,
   IndefiniteLengthBlockData = 1,
   RawLittleEndian = 2,
   RawBigEndian = 3
}
```

OBSERVATION 5.4.1

The `BinaryEncoding` enumeration indicates, for formatted I/O operations, the default format of binary data used in formatted I/O. The formats include IEEE definite and indefinite blocks and raw binary data with little or big endian byte ordering. The following table describes the enumeration members.

Value	Description
`DefiniteLengthBlockData`	IEEE-488.2 definite block format.
`IndefiniteLengthBlockData`	IEEE-488.2 indefinite block format.
`RawLittleEndian`	Raw binary data with little endian byte order.
`RawBigEndian`	Raw binary data with big endian byte order.

5.5. ByteOrder

DEFINITION

```
public enum ByteOrder
{
   BigEndian = 0,
   LittleEndian = 1
}
```

OBSERVATION 5.5.1

The `ByteOrder` enumeration indicates the byte order used in various VXI operations. The `ByteOrder` enumeration corresponds to the defined values for VISA's `VI_ATTR_SRC_BYTE_ORDER`, `VI_ATTR_DEST_BYTE_ORDER`, and `VI_ATTR_WIN_BYTE_ORDER` attributes.

5.6. DataWidth

DEFINITION

```
public enum DataWidth
{
   Width8 = 0,
   Width16 = 1,
   Width32 = 2,
   Width64 = 3
}
```

OBSERVATION 5.6.1

The `DataWidth` enumeration indicates the data width for register-based data transfer operations. This enumeration corresponds to the defined values for the source and destination width parameters in VISA's `viMove` function.

5.7. EventQueueStatus

DEFINITION

```
enum EventQueueStatus
{
   Empty = 0,
   NotEmpty = 1,
   Overflowed = 2

}
```

OBSERVATION 5.7.1

The `EventQueueStatus` enumeration indicates the current state of the event queue. The values include empty, not empty, and overflowed. Enumeration values are described in the following table.

Value	Description
`Empty`	The event queue is empty.
`NotEmpty`	The event queue is not empty.
`Overflowed`	The event queue has overflowed.

These correspond to the success status codes from VISA's `viWaitOnEvent` function.

5.8. EventType

DEFINITION

```
public enum EventType
{
   Custom = 0,
   AllEnabled = 1,
   ServiceRequest = 2,
   Clear = 3,
   GpibControllerInCharge = 4,
   GpibTalk = 5,
   GpibListen = 6,
   VxiVmeSystemFailure = 7,
   VxiVmeSystemReset = 8,
   VxiSignalProcessor = 9,
   VxiVmeInterrupt = 10,
   PxiInterrupt = 11,
   UsbInterrupt = 12,
   Trigger = 13
}
```

Refer to section 7.1, *Hardware Event APIs* for more information regarding how VISA events map to VISA.NET event types and events.

5.9. GpibAddressedState

DEFINITION

```
public enum GpibAddressedState
{
   Unaddressed = 0,
   Talker = 1,
   Listener = 2
}
```

OBSERVATION 5.9.1

The `GpibAddressedState` enumeration indicates whether the GPIB interface is currently addressed to talk or listen, or is not addressed. This enumeration corresponds to the defined values for the VISA `VI_ATTR_GPIB_ADDR_STATE` attribute.

5.10. GpibInstrumentRemoteLocalMode

DEFINITION

```
public enum GpibInstrumentRemoteLocalMode
{
   DeassertRen = 0,
   AssertRen = 1,
   GoToLocalDeassertRen = 2,
   AddressDeviceAssertRen = 3,
   AddressDeviceSendLocalLockout = 4,
   GoToLocal = 5
}
```

OBSERVATION 5.10.1

The `GpibInstrumentRemoteLocalMode` enumeration indicates the action to be taken by the SendRemoteLocalCommand of a GPIB INSTR session. This enumeration corresponds to defined values for the `mode` parameter of VISA's `viGpibControlREN` function, as shown in the table below. Values that are not relevant for GPIB instrument sessions are not included in the VISA.NET enumeration.

Enumeration Member	**VISA C Defined Value**
`DeassertRen`	`VI_GPIB_REN_DEASSERT`
`AssertRen`	`VI_GPIB_REN_ASSERT`
`GoToLocalDeassertRen`	`VI_GPIB_REN_DEASSERT_GTL`
`AddressDeviceAssertRen`	`VI_GPIB_REN_ASSERT_ADDRESS`
`AddressDeviceSendLocalLockout`	`VI_GPIB_REN_ASSERT_ADDRESS_LLO`
`GoToLocal`	`VI_GPIB_REN_ADDRESS_GTL`

5.11. GpibInterfaceRemoteLocalMode

DEFINITION

```
public enum GpibInterfaceRemoteLocalMode
{
   DeassertRen = 0,
   AssertRen = 1,
   LocalLockoutAssertRen = 2
}
```

OBSERVATION 5.11.1

The `GpibInterfaceRemoteLocalMode` enumeration indicates the action to be taken by the SendRemoteLocalCommand of a GPIB INTFC session. This enumeration corresponds to the defined values for the `mode` parameter of VISA's `viGpibControlREN` function, as shown in the table below. Values that are not relevant for GPIB interface sessions are not included in the VISA.NET enumeration.

Enumeration Member	**VISA C Defined Value**
`DeassertRen`	`VI_GPIB_REN_DEASSERT`
`AssertRen`	`VI_GPIB_REN_ASSERT`
`LocalLockoutAssertRen`	`VI_GPIB_REN_ASSERT_LLO`

5.12. HardwareInterfaceType

DEFINITION

```
public enum HardwareInterfaceType
{
   Custom = 0,
   Gpib = 1,
   Vxi = 2,
   GpibVxi = 3,
   Serial = 4,
   Pxi = 5,
   Tcp = 6,
   Usb = 7
};
```

OBSERVATION 5.12.1

The `HardwareInterfaceType` enumeration indicates the hardware interface type of the current session. This enumeration corresponds to the defined values for `VI_ATTR_INTF_TYPE`. The value `Custom` has been added to allow for vendor-specific types.

5.13. IOBuffers

DEFINITION

```
[Flags]
public enum IOBuffers
{
   Read = 1,
   Write = 2,
   ReadWrite = Read | Write
}
```

OBSERVATION 5.13.1

The `IOBuffers` enumeration indicates buffer(s) in the low-level I/O interface.This enumeration roughly corresponds to two defined values (`VI_IO_IN_BUF` and `VI_IO_OUT_BUF`) for the `mask` parameters in VISA's `viSetBuf` and `viFlush` functions. Note that in VISA.NET, this enumeration is not used for formatted I/O buffers.

5.14. IOProtocol

DEFINITION

```
public enum IOProtocol
{
   Normal = 0,
   Fdc = 1,
   HS488 = 2,
   Ieee4882 = 3,
   UsbTmcVendor = 4
}
```

OBSERVATION 5.14.1

The `IOProtocol` enumeration indicates which protocol to use on a particular session. Choices are dependent on the session. This enumeration corresponds to the defined values for VISA's `VI_ATTR_IO_PROT` attribute.

5.15. LineState

DEFINITION

```
public enum LineState
{
   Unknown = -1,
   Unasserted = 0,
   Asserted = 1,
}
```

OBSERVATION 5.15.1

The `LineState` enumeration indicates whether the line is asserted or not, or if the state is unknown. This enumeration corresponds to the defined values for several VISA attributes that describe line state as asserted or not asserted, including:

```
VI_ATTR_GPIB_REN_STATE
VI_ATTR_GPIB_ATN_STATE
VI_ATTR_GPIB_NDAC_STATE
VI_ATTR_GPIB_SRQ_STATE
VI_ATTR_ASRL_CTS_STATE
VI_ATTR_ASRL_DCD_STATE
VI_ATTR_ASRL_DSR_STATE
VI_ATTR_ASRL_DTR_STATE
VI_ATTR_ASRL_RI_STATE
VI_ATTR_ASRL_RTS_STATE
VI_ATTR_VXI_VME_SYSFAIL_STATE
```

5.16. NativeVisaAttribute

DEFINITION

```
public enum NativeVisaAttribute : uint
{
   AllowDma = 0x3fff001e,
   AllowWriteCombining = 0x3fff0246,
   AsyncReturnCount32 = 0x3fff4026,
   AsyncReturnCount64 = 0x3fff4028,
   CommanderLogicalAddress = 0x3fff006b,
   DestinationAccess = 0x3fff0039,
   DestinationByteOrder = 0x3fff003a,
   DestinationIncrement = 0x3fff0041,
   DeviceStatusByte = 0x3fff0189,

   EventType = 0x3fff4010,
   FastDataChannel = 0x3fff000d,
   FastDataChannelMode = 0x3fff000f,
   FastDataChannelUsePair = 0x3fff0013,
   FileAppendEnabled = 0x3fff0192,
   GpibAddressedState = 0x3fff005c,
   GpibAtnState = 0x3fff0057,
   GpibHS488CableLength = 0x3fff0069,
   GpibIsControllerInCharge = 0x3fff005e,
   GpibIsSystemController = 0x3fff0068,
   GpibNdacState = 0x3fff0062,
   GpibPrimaryAddress = 0x3fff0172,
   GpibRepeatAddressingEnabled = 0x3fff001b,
   GpibReceivedIsControllerInCharge = 0x3fff4193,
   GpibRenState = 0x3fff0181,
   GpibSecondaryAddress = 0x3fff0173,
   GpibSrqState = 0x3fff0067,
   GpibUnaddressEnabled = 0x3fff0184,
   Is4882Compliant = 0x3fff019f,
   ImmediateServant = 0x3fff0100,
   InterfaceName = 0xbfff00e9,
   InterfaceParentNumber = 0x3fff0101,
   InterfaceType = 0x3fff0171,
   InterfaceNumber = 0x3fff0176,
   IOProtocol = 0x3fff001c,
   JobId = 0x3fff4006,
   MainframeLogicalAddress = 0x3fff0070,
   ManufacturerId = 0x3fff00d9,
   ManufacturerName = 0xbfff0072,
   MaximumEventQueueLength = 0x3fff0005,
   MemoryBase32 = 0x3fff00ad,
   MemoryBase64 = 0x3fff00d0,
   MemorySize32 = 0x3fff00dd,
   MemorySize64 = 0x3fff00d1,
   MemorySpace = 0x3fff00de,
   ModelCode = 0x3fff00df,
```

```
ModelName = 0xbfff0077,
OperationName = 0xbfff4042,
PxiActualLinkWidth = 0x3fff0243,
PxiBackplaneDestinationTriggerBus = 0x3fff020e,
PxiBackplaneSourceTriggerBus = 0x3fff020d,
PxiBusNumber = 0x3fff0205,
PxiChassis = 0x3fff0206,
PxiDeviceNumber = 0x3fff0201,
PxiDStarBus = 0x3fff0244,
PxiDStarSet = 0x3fff0245,
PxiFunctionNumber = 0x3fff0202,
PxiIsExpress = 0x3fff0240,
PxiMaximumLinkWidth = 0x3fff0242,
PxiMemoryBase32Bar0 = 0x3fff0221,
PxiMemoryBase32Bar1 = 0x3fff0222,
PxiMemoryBase32Bar2 = 0x3fff0223,
PxiMemoryBase32Bar3 = 0x3fff0224,
PxiMemoryBase32Bar4 = 0x3fff0225,
PxiMemoryBase32Bar5 = 0x3fff0226,
PxiMemoryBase64Bar0 = 0x3fff0228,
PxiMemoryBase64Bar1 = 0x3fff0229,
PxiMemoryBase64Bar2 = 0x3fff022a,
PxiMemoryBase64Bar3 = 0x3fff022b,
PxiMemoryBase64Bar4 = 0x3fff022c,
PxiMemoryBase64Bar5 = 0x3fff022d,
PxiMemorySize32Bar0 = 0x3fff0231,
PxiMemorySize32Bar1 = 0x3fff0232,
PxiMemorySize32Bar2 = 0x3fff0233,
PxiMemorySize32Bar3 = 0x3fff0234,
PxiMemorySize32Bar4 = 0x3fff0235,
PxiMemorySize32Bar5 = 0x3fff0236,
PxiMemorySize64Bar0 = 0x3fff0238,
PxiMemorySize64Bar1 = 0x3fff0239,
PxiMemorySize64Bar2 = 0x3fff023a,
PxiMemorySize64Bar3 = 0x3fff023b,
PxiMemorySize64Bar4 = 0x3fff023c,
PxiMemorySize64Bar5 = 0x3fff023d,
PxiMemoryTypeBar0 = 0x3fff0211,
PxiMemoryTypeBar1 = 0x3fff0212,
PxiMemoryTypeBar2 = 0x3fff0213,
PxiMemoryTypeBar3 = 0x3fff0214,
PxiMemoryTypeBar4 = 0x3fff0215,
PxiMemoryTypeBar5 = 0x3fff0216,
PxiReceivedInterruptData = 0x3fff4241,
PxiReceivedInterruptSequence = 0x3fff4240,
PxiSlotLinkWidth = 0x3fff0241,
PxiSlotLocalBusLeft = 0x3fff0208,
PxiSlotLocalBusRight = 0x3fff0209,
PxiSlotPath = 0xbfff0207,
PxiStarTriggerBus = 0x3fff020b,
PxiStarTriggerLine = 0x3fff020c,
PxiTriggerBus = 0x3fff020a,
ReadBufferOperationMode = 0x3fff002a,
```

```
        ReadBufferSize = 0x3fff002b,
        ReceivedInterruptLevel = 0x3fff4041,
        ReceivedInterruptStatusId = 0x3fff4023,
        ReceivedSignalProcessorStatusId = 0xbfff4011,
        ReceivedTcpAddress = 0xbfff4198,
        ReceivedTriggerId = 0x3fff4012,
        ResourceManagerSession = 0x3fff00c4,
        ResourceClass = 0xbfff0001,
        ResourceImplementationVersion = 0x3fff0003,
        ResourceLockState = 0x3fff0004,
        ResourceManufacturerId = 0x3fff0175,
        ResourceManufacturerName = 0xbfff0174,
        ResourceName = 0xbfff0002,
        ResourceSpecificationVersion = 0x3fff0170,
        SendEndEnabled = 0x3fff0016,
        SerialAvailableByteCount = 0x3fff00ac,
        SerialBaud = 0x3fff0021,
        SerialCtsState = 0x3fff00ae,
        SerialDataBits = 0x3fff0022,
        SerialDcdState = 0x3fff00af,
        SerialDsrState = 0x3FFF00b1,
        SerialDtrState = 0x3fff00b2,
        SerialEndIn = 0x3fff00b3,
        SerialEndOut = 0x3fff00b4,
        SerialFlowControl = 0x3fff0025,
        SerialParity = 0x3fff0023,
        SerialReplaceCharacter = 0x3fff00be,
        SerialRIState = 0x3fff00bf,
        SerialRtsState = 0x3fff00c0,
        SerialStopBits = 0x3fff0024,
        SerialXOffCharacter = 0x3fff00c2,
        SerialXOnCharacter = 0x3fff00c1,
        Slot = 0x3fff00e8,
        SourceAccess = 0x3fff003c,
        SourceByteOrder = 0x3fff003d,
        SourceIncrement = 0x3fff0040,
        Status = 0x3fff4025,
        SuppressEndEnabled = 0x3fff0036,
        TcpAddress = 0xbfff0195,
        TcpDeviceName = 0xbfff0199,
        TcpHiSLIPMaximumMessageSizeKB = 0x3fff0302,
        TcpHiSLIPOverlapEnabled = 0x3fff0300,
        TcpHiSLIPVersion = 0x3fff0301,
        TcpHostName = 0xbfff0196,
        TcpIsHiSLIP = 0x3fff0303,
        TcpKeepAlive = 0x3fff019b,
        TcpNoDelay = 0x3fff019a,
        TcpPort = 0x3fff0197,
        TerminationCharacter = 0x3fff0018,
        TerminationCharacterEnabled = 0x3fff0038,
        TimeoutValue = 0x3fff001a,
        TriggerId = 0x3fff0177,
        UsbInterfaceNumber = 0x3fff01a1,
```

```
    UsbMaximumInterruptSize = 0x3fff01af,
    UsbProtocol = 0x3fff01a7,
    UsbReceivedInterruptSize = 0x3fff41b0,
    UsbSerialNumber = 0xbfff01a0,
    UserData32 = 0x3fff0007,
    VxiDeviceClass = 0x3fff006c,
    VxiLogicalAddress = 0x3fff00d5,
    VxiTriggerStatus = 0x3fff008d,
    VxiTriggerSupport = 0x3fff0194,
    VxiVmeInterruptStatus = 0x3fff008b,
    VxiVmeSystemFailureState = 0x3fff0094,
    WindowAccess = 0x3fff00c3,
    WindowAccessPrivilege0x3fff0045,
    WindowBaseAddress32 = 0x3fff0098,
    WindowBaseAddress64 = 0x3fff009b,
    WindowByteOrder = 0x3fff0047,
    WindowSize32 = 0x3fff009a,
    WindowSize64 = 0x3fff009c,
    WriteBufferOperationMode = 0x3fff002d,
    WriteBufferSize = 0x3fff002e,
}
```

OBSERVATION 5.16.1

The `NativeVisaAttribute` enumeration corresponds to the defined values for the VISA attributes.

5.17. PxiMemoryType

DEFINITION

```
public enum PxiMemoryType
{
   None = 0,
   Memory = 1,
   IO = 2,
}
```

OBSERVATION 5.17.1

The `PxiMemoryType` enumeration indicates the memory type (memory mapped or I/O mapped) used by the device in the specified base address register (BAR). This enumeration corresponds to the defined values for the VISA attributes `VI_ATTR_PXI_MEM_TYPE_BAR`*n*.

5.18. ReadStatus

DEFINITION

```
public enum ReadStatus
{
   Unknown = 0,
   EndReceived = 1,
   TerminationCharacterEncountered = 2,
   MaximumCountReached = 3
}
```

OBSERVATION 5.18.1

The `ReadStatus` enumeration indicates the success status of a raw I/O read operation. This enumeration corresponds to the defined success status codes for VISA's `viRead` function but adds the Unknown member. The Unknown member is the default and is used for the initial state, but it will never be returned for a successful read operation.

5.19. RemoteLocalMode

DEFINITION

```
public enum RemoteLocalMode
{
   LocalWithoutLockout = 0,
   Remote = 1,
   RemoteWithLocalLockout = 2,
   Local = 3
}
```

OBSERVATION 5.19.1

The `RemoteLocalMode` enumeration indicates the action to be taken by the SendRemoteLocalCommand of a GPIB, TCPIP, or USB INSTR session. This enumeration corresponds to the defined values for the `mode` parameter of VISA's viGpibControlREN function, as shown in the table below. Values that are not relevant are not included in the VISA.NET enumeration.

Enumeration Member	**VISA C Defined Value**
`LocalWithoutLockout`	`VI_GPIB_REN_DEASSERT_GTL`
`Remote`	`VI_GPIB_REN_ASSERT_ADDRESS`
`RemoteWithLocalLockout`	`VI_GPIB_REN_ASSERT_ADDRESS_LLO`
`Local`	`VI_GPIB_REN_ADDRESS_GTL`

5.20. ResourceLockState

DEFINITION

```
public enum ResourceLockState
{
   NoLock = 0,
   ExclusiveLock = 1,
   SharedLock = 2
}
```

OBSERVATION 5.20.1

The `RemoteLocalMode` enumeration indicates the state of the VISA lock on the resource associated with this session. This enumeration corresponds to the defined values for the VISA attribute `VI_ATTR_RSRC_LOCK_STATE`.

5.21. ResourceOpenStatus

DEFINITION

```
public enum ResourceOpenStatus
{
   Success = 0,
   DeviceNotResponding = 1,
   ConfigurationNotLoaded = 2
}
```

OBSERVATION 5.21.1

The `ResourceOpenStatus` enumeration indicates the success status of an open operation. This enumeration corresponds to the defined success status codes for VISA's `viOpen` function.

5.22. SerialFlowControlModes

DEFINITION

```
[Flags]
public enum SerialFlowControlModes
{
   None = 0,
   XOnXOff = 1,
   RtsCts = 2,
   DtrDsr = 4
}
```

OBSERVATION 5.22.1

The `SerialFlowControlModes` enumeration indicates the type of flow control used by the Serial connection. This enumeration corresponds to the defined values for the VISA attribute `VI_ATTR_ASRL_FLOW_CNTRL`.

5.23. SerialParity

DEFINITION

```
public enum SerialParity
{
   None = 0,
   Odd = 1,
   Even = 2,
   Mark = 3,
   Space = 4
}
```

OBSERVATION 5.23.1

The `SerialParity` enumeration indicates whether parity checking is being used by the serial connection, and if so, how it is determined. The specified parity is used with every frame transmitted and received. This enumeration corresponds to the defined values for the VISA attribute `VI_ATTR_ASRL_PARITY`.

5.24. SerialTerminationMethod

DEFINITION

```
public enum SerialTerminationMethod
{
   None = 0,
   HighestBit = 1,
   TerminationCharacter = 2,
   Break = 3
}
```

OBSERVATION 5.24.1

The `SerialTermination` enumeration indicates the method used to terminate Serial read and write operations. This enumeration corresponds to the defined values for the VISA attributes `VI_ATTR_ASRL_END_IN` and `VI_ATTR_ASRL_END_OUT`.

5.25. StatusByteFlags

DEFINITION

```
[Flags]
public enum StatusByteFlags : short
{
   User0 = 0x01,
   User1 = 0x02,
   User2 = 0x04,
   User3 = 0x08,
   MessageAvailable = 0x10,
   EventStatusRegister = 0x20,
   RequestingService = 0x40,
   User7 = 0x80
}
```

OBSERVATION 5.25.1

The `StatusByteFlags` enumeration indicates individual bits of the IEEE 488.2 Status Byte. This enumeration allows possible values for the VISA attribute `VI_ATTR_DEV_STATUS_BYTE` to be expressed as a combination of the enumeration values.

5.26. SerialStopBitsMode

DEFINITION

```
public enum SerialStopBitsMode
{
   One = 0,
   OneAndOneHalf = 1,
   Two = 2
}
```

OBSERVATION 5.26.1

The `SerialStopBitsMode` enumeration indicates the number of stop bits used to indicate the end of a Serial frame. This enumeration corresponds to the defined values for the VISA attribute `VI_ATTR_ASRL_STOP_BITS`.

5.27. TriggerLine

DEFINITION

```
public enum TriggerLine
{
   All = -2,
   Ttl0 = 0,
   Ttl1 = 1,
   Ttl2 = 2,
   Ttl3 = 3,
   Ttl4 = 4,
   Ttl5 = 5,
   Ttl6 = 6,
   Ttl7 = 7,
   Ecl0 = 8,
   Ecl1 = 9,
   Ecl2 = 10,
   Ecl3 = 11,
   Ecl4 = 12,
   Ecl5 = 13,
   StarSlot1 = 14,
   StarSlot2 = 15,
   StarSlot3 = 16,
   StarSlot4 = 17,
   StarSlot5 = 18,
   StarSlot6 = 19,
   StarSlot7 = 20,
   StarSlot8 = 21,
   StarSlot9 = 22,
   StarSlot10 = 23,
   StarSlot11 = 24,
   StarSlot12 = 25,
   StarInstrument = 26,
   PanelIn = 27,
   PanelOut = 28,
   StarVxi0 = 29,
   StarVxi1 = 30,
   StarVxi2 = 31,
   Ttl8 = 32,
   Ttl9 = 33,
   Ttl10 = 34,
   Ttl11 = 35
}
```

OBSERVATION 5.27.1

The `TriggerLine` enumeration indicates a VXI or PXI trigger line. This enumeration corresponds to the defined values for VISA triggers. The defined values for VISA triggers include values that begin with `VI_TRIG_`, except for VI_TRIG_SW and values that begin with `VI_TRIG_PROT_`.

5.28. TriggerLines

DEFINITION

```
[Flags]
public enum TriggerLines
{
   Ecl0 = 1 << TriggerLine.Ecl0,
   Ecl1 = 1 << TriggerLine.Ecl1,
   Ecl1 = 1 << TriggerLine.Ecl2,
   Ecl1 = 1 << TriggerLine.Ecl3,
   Ecl1 = 1 << TriggerLine.Ecl4,
   Ecl1 = 1 << TriggerLine.Ecl5,
   PanelIn = 1 << TriggerLine.PanelIn,
   PanelOut = 1 << TriggerLine.PanelOut,
   StarInstr = 1 << TriggerLine.StarInstrument,
   StarSlot1 = 1 << TriggerLine.StarSlot1,
   StarSlot2 = 1 << TriggerLine.StarSlot2,
   StarSlot3 = 1 << TriggerLine.StarSlot3,
   StarSlot4 = 1 << TriggerLine.StarSlot4,
   StarSlot5 = 1 << TriggerLine.StarSlot5,
   StarSlot6 = 1 << TriggerLine.StarSlot6,
   StarSlot7 = 1 << TriggerLine.StarSlot7,
   StarSlot8 = 1 << TriggerLine.StarSlot8,
   StarSlot9 = 1 << TriggerLine.StarSlot9,
   StarSlot10 = 1 << TriggerLine.StarSlot10,
   StarSlot11 = 1 << TriggerLine.StarSlot11,
   StarSlot12 = 1 << TriggerLine.StarSlot12,
   StarVxi0 = 1 << TriggerLine.StarVxi0,
   StarVxi1 = 1 << TriggerLine.StarVxi1,
   StarVxi2 = 1 << TriggerLine.StarVxi2,
   Ttl0 = 1 << TriggerLine.Ttl0,
   Ttl1 = 1 << TriggerLine.Ttl1,
   Ttl2 = 1 << TriggerLine.Ttl2,
   Ttl3 = 1 << TriggerLine.Ttl3,
   Ttl4 = 1 << TriggerLine.Ttl4,
   Ttl5 = 1 << TriggerLine.Ttl5,
   Ttl6 = 1 << TriggerLine.Ttl6,
   Ttl7 = 1 << TriggerLine.Ttl7
}
```

OBSERVATION 5.28.1

The `TriggerLines` enumeration indicates one or more VXI trigger lines. This enumeration corresponds to the defined values for VISA triggers. The defined values for VISA triggers include values that begin with `VI_TRIG_`, except for VI_TRIG_SW and values that begin with `VI_TRIG_PROT_`. TTL lines 8-11 are not included, as they apply to PXI only.

5.29. VxiAccessPriviledge

DEFINITION

```
public enum VxiAccessPrivilege
{
   DataPrivileged = 0,
   DataNonPrivileged = 1,
   ProgramPrivileged = 2,
   ProgramNonPrivileged = 3,
   BlockPrivileged = 4,
   BlockNonPrivileged = 5,
   D64Privileged = 6,
   D64NonPrivileged = 7
   D64DoubleEdgeVme = 8,
   D64Sst160 = 9,
   D64Sst267 = 10,
   D64Sst320 = 11
 }
```

OBSERVATION 5.29.1

The `VxiAccessPriviledge` enumeration indicates the address modifier to be used in high-level access operations when writing to the destination. This enumeration corresponds to the defined values for the VISA attributes `VI_ATTR_SRC_ACCESS_PRIV` and `VI_ATTR_DEST_ACCESS_PRIV`.

5.30. VxiCommandMode

DEFINITION

```
public enum VxiCommandMode
{
   Command16Bit = 0,
   Command32Bit = 1,
   Command32BitResponse16Bit = 2,
   CommandResponse16Bit = 3,
   CommandResponse32Bit = 4,
   Response16Bit = 5,
   Response32Bit = 6
}
```

OBSERVATION 5.30.1

The `VxiCommandMode` enumeration indicates whether to VISA should issue a command and/or retrieve a response, and what type or size of command and/or response to use. This enumeration corresponds to the defined values for the `mode` parameter of VISA's `viVxiCommandQuery` function.

5.31. VxiDeviceClass

DEFINITION

```
public enum VxiDeviceClass
{
   Memory = 0,
   Extended = 1,
   Message = 2,
   Register = 3,
   Other = 4
}
```

OBSERVATION 5.31.1

The `VxiDeviceClass` enumeration indicates the VXI-defined device class to which a particular resource belongs. This enumeration corresponds to the defined values for the VISA attribute `VI_ATTR_VXI_DEV_CLASS`.

5.32. VxiTriggerProtocol

DEFINITION

```
public enum VxiTriggerProtocol
{
   Software = 0,
   On = 1,
   Off = 2,
   Sync = 5,
}
```

OBSERVATION 5.32.1

The `VxiTriggerProtocol` enumeration indicates the trigger protocol to be used when a VXI trigger is asserted. This enumeration corresponds to the defined values for the `protocol` parameter of VISA's `viAssertTrigger` function, although the Software member corresponds to the case where the VISA `VI_ATTR_TRIG_ID` is set to `VI_TRIG_SW`.

5.33. VxiUtilitySignal

DEFINITION

```
public enum VxiUtilitySignal
{
   AssertSystemReset = 0,
   AssertSystemFailure = 1,
   DeassertSystemFailure = 2,
}
```

OBSERVATION 5.33.1

The `VxiUtilitySignal` enumeration indicates the utility bus signal to assert. This is valid only for VXI BACKPLANE sessions. This enumeration corresponds to the defined values for the `line` parameter of VISA's `viAssertUtilSignal` function.

Section 6: VISA.NET Exceptions and Status Codes

In general, VISA.NET implementations are free to throw applicable exceptions when needed. There are just a few special cases where particular exceptions are required for specific error conditions in specific methods or properties.

6.1. Exception Overview

The .NET Framework has a rich list of exceptions and guidelines for using them. Most .NET programmers will expect these exceptions to be used when they are appropriate. In cases where the exception is specific to VISA.NET, the exception should either be `VisaException` or derived from `VisaException`. If the VISA.NET implementation overlays a native VISA implementation, and the VISA implementation returns an error status code, the VISA.NET exception should be `NativeVisaException`.

VISA.NET defines the following exceptions. All exceptions are defined in the Ivi.Visa namespace.

- `VisaException`
- `VisaIoTimeoutException`
- `NativeVisaException`
- `TypeFormatterException`

`NativeVisaException` is specifically for reporting errors from an underlying VISA C implementation. This exception includes the error status code reported by VISA C. VISA.NET includes a class of error status codes, `NativeErrorCode`, that enables calling programs to use a convenient name for errors rather than a number.

All exceptions defined by VISA.NET derive from `VisaException`.

Since calling programs routinely need to handle I/O timeout exceptions, there are some specific rules and observations related to throwing timeout exceptions.

In cases where VISA C would return an error code, the corresponding VISA.NET method or property is expected to throw an exception unless otherwise specified.

I/O TIMEOUT EXCEPTIONS

RULE 6.1.1

Certain methods specify that `Ivi.Visa.IoTimeoutException` **SHALL** be thrown when an I/O operation times out. Whenever a VISA.NET I/O timeout is reported by one of these methods, it **SHALL** be reported with `Ivi.Visa.IoTimeoutException`, regardless of whether the underlying implementation delegates to VISA C, or is a native .NET implementation. In these cases in particular, it **SHALL NOT** be reported using `System.TimeoutException`, `Ivi.Visa.NativeVisaException`, or any other exception that might otherwise look suitable.

OBSERVATION 6.1.1

In cases where it is specified that `Ivi.Visa.IoTimeoutException` shall be thrown to report a timeout condition, calling programs may reliably expect that exception to be thrown when an I/O timeout occurs.

PERMISSION 6.1.1

Methods that do not explicitly specify that `Ivi.Visa.IoTimeoutException` shall be thrown to report a timeout condition, may throw `System.TimeoutException`, `Ivi.Visa.NativeVisaException`, or any other exception that might be suitable to report the timeout.

OTHER EXCEPTIONS

RULE 6.1.2

If a VISA.NET I/O implementation throws `Ivi.Visa.NativeVisaException` or any exception that derives from `Ivi.Visa.NativeVisaException` to report an error that was returned by the underlying

VISA C implementation, that exception's `StatusCode` property **SHALL** match the value of the status code returned by VISA C.

RULE 6.1.3

A VISA.NET I/O implementation **SHALL NOT** throw `Ivi.Visa.NativeVisaException` or any exception that derives from `Ivi.Visa.NativeVisaException` unless the VISA.NET implementation is based on an underlying VISA C implementation.

PERMISSION 6.1.2

Except as noted in RULE 6.1.1, if a VISA.NET I/O session's implementation is layered over a VISA C implementation, any operation may throw an `Ivi.Visa.NativeVisaException` with a vendor specific status code that is not listed in the VISA C specifications.

PERMISSION 6.1.3

VISA.NET implementations may define vendor specific VISA.NET exceptions.

RULE 6.1.4

Vendor defined VISA.NET exceptions **SHALL** derive from `VisaException` directly or indirectly, as appropriate.

RECOMMENDATION 6.1.2

Vendors should not create a vendor specific VISA.NET exception if there is an applicable .NET framework exception. For example vendors should not define vendor specific VISA.NET exceptions to replace `System.ArgumentNullException` or `System.ArgumentOutOfRangeException`.

PERMISSION 6.1.4

Except as noted in RULE 6.1.1, vendor specific VISA.NET implementations may allow exceptions thrown by the .NET Framework to propagate up to the calling program.

OBSERVATION 6.1.2

Any VISA.NET operation may throw exceptions, particularly .NET Framework exceptions or vendor-specific exceptions, not listed in this specification.

OBSERVATION 6.1.3

In light of the previous two permissions, it is important that calling programs (1) follow .NET guidelines for handling exceptions, (2) not assume that particular exceptions will be returned for a particular error condition (except as noted in RULE 6.1.1), since different vendors may return different errors in the same situation, and (3) not restrict error processing to VISA status codes defined by the VISA sepcifications when an `Ivi.Visa.NativeVisaException` is caught.

OBSERVATION 6.1.4

`Ivi.Visa.NativeVisaException` is only thrown when the underlying implementation delegates to a VISA C implementation.

RULE 6.1.5

A VISA.NET I/O session **SHALL NOT** throw `System.NotImplementedException`.

6.2. VISA.NET Exceptions

6.2.1. Ivi.Visa.VisaException

DESCRIPTION

A VISA.NET error has occurred.

DEFINITION

```
public class VisaException : System.Exception
{
   public VisaException(){…}
   public VisaException(String message) {…}
   public VisaException(String message,
                        System.Exception innerException) {…}
   protected VisaException(SerializationInfo info,
                           StreamingContext context) {…}
}
```

DEFAULT MESSAGE STRING

```
Exception of type 'Ivi.Visa.VisaException' was thrown.
```

ARGUMENTS

Name	Description	Base Type
`message`	A message appropriate to the error being reported.	`System.String`
`innerException`	If not null, the exception that is the cause of the current exception.	`System.Exception` or derived type

IMPLEMENTATION NOTES

`VisaException` is implemented in the VISA.NET standard components.

6.2.2. Ivi.Visa.IOTimeoutException

DESCRIPTION

A VISA.NET I/O timeout has occured.

DEFINITION

```
public class Ivi.Visa.IOTimeoutException : Ivi.Visa.VisaException
{
   public IOTimeoutException (Int64 actualCount, Byte[] actualData) {…}
   public IOTimeoutException (Int64 actualCount, Byte[] actualData,
                              String message) {…}
   public IOTimeoutException (Int64 actualCount, Byte[] actualData,
                              String message,
                              System.Exception innerException) {…}
   protected IOTimeoutException (SerializationInfo info,
                                      StreamingContext context) {…}
   public Int64 ActualCount { get; protected set; }
   public Byte[] ActualData { get; protected set; }
}
```

DEFAULT MESSAGE STRING

```
Exception of type 'Ivi.Visa.IOTimeoutException' was thrown.
```

ARGUMENTS

Name	Description	Base Type
`actualCount`	The actual number of elements read or written before the timeout occurred. A value of -1 indicates that the actual number could not be determined.	`System.Int64`
`actualData`	The actual bytes read or written before the timeout occurred. If the actual number of elements read could not be determined, the array is empty.	`System.Byte[]`
`message`	A message appropriate to the error being reported.	`System.String`
`innerException`	The exception that is the cause of the current exception. If the innerException parameter is not null, the current exception is raised in a catch block that handles the inner exception.	`System.Exception` or derived type

PROPERTIES

Name	Description	Base Type
`ActualCount`	The actual number of elements read or written before the timeout occurred. A value of -1 indicates that the actual number could not be determined.	`System.Int64`
`ActualData`	The actual bytes read or written before the timeout occurred. If the actual number of elements read could not be determined, the array is empty.	`System.Byte[]`

IMPLEMENTATION NOTES

`IOTimeoutException` is implemented in the VISA.NET standard components.

6.2.3. Ivi.Visa.NativeVisaException

DESCRIPTION

An error related to the underlying VISA native C implementation has occurred. The status code indicates the type of error that occurred.

DEFINITION

```
public class NativeVisaException : Ivi.Visa.VisaException
{
   public NativeVisaException(int errorCode) {…}
   public NativeVisaException(int errorCode, String message) {…}
   public NativeVisaException(int errorCode, String message,
                              System.Exception innerException) {…}
   protected NativeVisaException(SerializationInfo info,
                                 StreamingContext context) {…}
   public int ErrorCode { get; protected set; }
}
```

DEFAULT MESSAGE STRING

```
Exception of type 'Ivi.Visa.NativeVisaException' was thrown.
```

ARGUMENTS

Name	Description	Base Type
`errorCode`	The underlying VISA status code of the error that occurred.	`System.Int32`
`message`	A message appropriate to the error being reported.	`System.String`
`innerException`	The exception that is the cause of the current exception. If the innerException parameter is not null, the current exception is raised in a catch block that handles the inner exception.	`System.Exception` or derived type

PROPERTIES

Name	Description	Base Type
`ErrorCode`	The underlying VISA status code of the error that occurred.	`System.Int32`

IMPLEMENTATION NOTES

`NativeVisaException` is implemented in the VISA.NET standard components.

6.2.4. Ivi.Visa.TypeFormatterException

DESCRIPTION

A Type Formatter error has occurred. This could be an error either in converting the type value to a string, or in converting a string to the corresponding type value.

A type formatter is a class that implements the `ITypeFormatter` interface, which is used by Printf and Scanf methods to format the values of arbitrary types. Refer to 9.3.2, ITypeFormatter Interface for more information.

Type formatter exceptions are intended to be thrown by classes that implement `ITypeFormatter`.

DEFINITION

```
public class TypeFormatterException : System.Exception
{
   public TypeFormatterException(){…}
   public TypeFormatterException(System.Exception innerException) {…}

   public TypeFormatterException(Type type) {…}
   public TypeFormatterException(Type type,
                                 System.Exception innerException) {…}
   public TypeFormatterException(Type type,
                                 String instrumentResponse) {…}
   public TypeFormatterException(Type type,
                                 String instrumentResponse,
                                 System.Exception innerException) {…}

   public TypeFormatterException(Object obj) {…}
   public TypeFormatterException(Object obj,
                                 System.Exception innerException) {…}

   protected TypeFormatterException(SerializationInfo info,
                                    StreamingContext context) {…}
}
```

DEFAULT MESSAGE STRING

```
Exception of type 'Ivi.Visa.TypeFormatterException was thrown.
```

ARGUMENTS

Name	Description	Base Type
`type`	The type of the object value being formatted or parsed.	`System.Type`
`instrumentResponse`	The instrument response whose format could not be correctly parsed by the type formatter.	`System.String`
`obj`	The object whose value is being formatted.	`System.Object`
`args`	The collection of objects to be used in formatting the message.	`System.Object[]`

`message`	A message appropriate to the error being reported. For the two constructors with `args` arguments, the message is a format string capable of formatting the accompanying arguments.	`StatusCodes`
`innerException`	The exception that is the cause of the current exception. If the innerException parameter is not null, the current exception is raised in a catch block that handles the inner exception.	`System.Exception` or derived type

IMPLEMENTATION NOTES

OBSERVATION 6.2.1

`TypeFormatterException` is intended for use by objects that implement the `ITypeFormatter` interface. Note that while the `ITypeFormatter` interface and `TypeFormatterException` are defined in the VISA.NET Shared Components, objects that implement the `ITypeFormatter` interface are not provided.

OBSERVATION 6.2.2

The following guidelines are provided for selecting an appropriate constructor when throwing `TypeFormatterException` from an object that implements `ITypeFormatter`.

- The first two constructors in the above list, and the last, will typically not be used.
- The two constructors that take a `type` argument without the `instrumentResponse` argument, and the two constructors that take an `obj` argument, are typically used to throw `Printf` formatting errors.
- The two constructors that take a `type` argument with the `instrumentResponse` argument are typically used to throw `Scanf` parsing errors.

6.3. NativeErrorCode Class

DESCRIPTION

The NativeErrorCode class consists of constants for all of the standard error status codes that are defined in the VISA C specification. This class is provided for convenience when using NativeVisaException. VISA success and warning status codes are not included in the `NativeErrorCode` class.

The `NativeErrorCode` class includes one method that returns the VISA C constant name of the error code, with the leading "VI_" removed.

DEFINITION

```
public class NativeErrorCode
{
   public const int SystemError = -1073807360;
   public const int InvalidObject = -1073807346;
   public const int ResourceLocked = -1073807345;
   public const int InvalidExpression = -1073807344;
   public const int ResourceNotFound = -1073807343;
   public const int InvalidResourceName = -1073807342;
   public const int InvalidAccessMode = -1073807341;
   public const int Timeout = -1073807339;
   public const int CloseFailed = -1073807338;
   public const int InvalidDegree = -1073807333;
   public const int InvalidJobId = -1073807332;
   public const int UnsupportedAttribute = -1073807331;
   public const int UnsupportedAttributeValue = -1073807330;
   public const int ReadOnlyAttribute = -1073807329;
   public const int InvalidLockType = -1073807328;
   public const int InvalidAccessKey = -1073807327;
   public const int InvalidEvent = -1073807322;
   public const int InvalidMechanism = -1073807321;
   public const int HandlerNotInstalled = -1073807320;
   public const int InvalidHandlerReference = -1073807319;
   public const int InvalidEventContext = -1073807318;
   public const int QueueOverflow = -1073807315;
   public const int NotEnabled = -1073807313;
   public const int Abort = -1073807312;
   public const int RawWriteProtocolViolation = -1073807308;
   public const int RawReadProtocolViolation = -1073807307;
   public const int OutputProtocolViolation = -1073807306;
   public const int InputProtocolViolation = -1073807305;
   public const int BusError = -1073807304;
   public const int OperationInProgress = -1073807303;
   public const int InvalidSetup = -1073807302;
   public const int QueueError = -1073807301;
   public const int MemoryAllocation = -1073807300;
   public const int InvalidBufferMask = -1073807299;
   public const int IOError = -1073807298;
   public const int InvalidFormatSpecifier = -1073807297;
   public const int UnsupportedFormatSpecifier = -1073807295;
   public const int TriggerLineInUse = -1073807294;
   public const int TriggerLineNotReserved = -1073807293;
```

```
        public const int UnsupportedMode = -1073807290;
        public const int ServiceRequestNotReceived = -1073807286;
        public const int InvalidAddressSpace = -1073807282;
        public const int InvalidOffset = -1073807279;
        public const int InvalidDataWidth = -1073807278;
        public const int UnsupportedOffset = -1073807276;
        public const int VariableDataWidthNotSupported = -1073807275;
        public const int WindowNotMapped = -1073807273;
        public const int ResponsePending = -1073807271;
        public const int NoListeners = -1073807265;
        public const int NotControllerInCharge = -1073807264;
        public const int NotSystemController = -1073807263;
        public const int OperationNotSupported = -1073807257;
        public const int InterruptPending = -1073807256;
        public const int ParityError = -1073807254;
        public const int FramingError = -1073807253;
        public const int Overrun = -1073807252;
        public const int TriggerNotMapped = -1073807250;
        public const int OffsetNotAligned = -1073807248;
        public const int UserBufferError = -1073807247;
        public const int ResourceBusy = -1073807246;
        public const int WidthNotSupported = -1073807242;
        public const int InvalidParameter = -1073807240;
        public const int InvalidProtocol = -1073807239;
        public const int InvalidWindowSize = -1073807237;
        public const int WindowAlreadyMapped = -1073807232;
        public const int OperationNotImplemented = -1073807231;
        public const int InvalidLength = -1073807229;
        public const int InvalidMode = -1073807215;
        public const int SessionNotLocked = -1073807204;
        public const int MemoryNotShared = -1073807203;
        public const int LibraryNotFound = -1073807202;
        public const int UnsupportedInterrupt = -1073807201;
        public const int InvalidLine = -1073807200;
        public const int FileAccessError = -1073807199;
        public const int FileIOError = -1073807198;
        public const int TriggerLineNotSupported = -1073807197;
        public const int EventMechanismNotSupported = -1073807196;
        public const int InterfaceNumberNotConfigured = -1073807195;
        public const int ConnectionLost = -1073807194;
        public const int MachineNotAvailable = -1073807193;
        public const int AccessDenied = -1073807192;
        public const int ServerCertificateError = -1073807184;
        public const int ServerCertificateUntrusted = -1073807183;
        public const int ServerCertificateExpired = -1073807182;
        public const int ServerCertificateRevoked = -1073807181;
        public const int ServerCertificateInvalidSubject = -1073807180;

        public static string GetMacroNameFromStatusCode(int status) {…}
    }
```

IMPLEMENTATION NOTES

The `NativeErrorCode` class is implemented in the VISA.NET standard components.

6.3.1. GetMacroNameFromStatusCode()

DESCRIPTION

Given a VISA C error status code, this method returns the name of the VISA C defined constant with the leading "VI_" removed.

DEFINITION

```
static String GetMacroNameFromStatusCode(Int32 status) {…}
```

ARGUMENTS

Name	Description	Base Type
`status`	The error status code.	`System.Int32`
`Return Value`	The name of the VISA C defined constant with the leading "VI_" removed	`System.String`

Section 7: VISA.NET Hardware Events

VISA.NET *hardware events* are used by VISA.NET sessions to report things that the calling program may need to know about. For the most part, these events are related to the I/O hardware associated with the session – interrupts, service requests, triggers, and so on.

(These events are designated *hardware events* to distinguish them from notifications connected with asynchronous I/O. Asynchronous I/O is described in Section 9.2.2, *Asynchronous I/O*, along with associated notification mechanisms.)

7.1. Hardware Event APIs

VISA.NET handles hardware events two different ways. The first maps very closely to the functions that VISA C uses for events. The second takes advantage of .NET events.

HARDWARE EVENT METHODS

VISA C provides a number of functions that enable calling programs to register for and receive notification of hardware events. Each of these functions has an event type parameter that identifies one or more events.

- `viEnableEvent()` enables the event, so that an event will be "fired" when the corresponding condition is detected.
- `viDisableEvent()` disables the event, so that it is never fired.
- `viDiscardEvent()` discards events from the event queue. (If events happen more quickly than they can be handled, they are added to a queue until they can be handled.)
- `viWaitOnEvent()` waits for an event of the specified event type(s) to be fired.

VISA.NET provides methods that correspond to the VISA C methods. The VISA.NET `EnableEvent()` method corresponds to `viEnableEvent()` called with the event mechanism of `VI_QUEUE`. The `DisableEvent()`, `DiscardEvent()`, and `WaitOnEvent()` methods correspond exactly to the corresponding VISA C functions. Each method includes an `eventType` argument that indicates the kind of event to which the method applies. These methods are part of `IVisaSession`, and so are included with every type of VISA.NET session.

.NET HARDWARE EVENTS

The second way that VISA.NET handles hardware events is with VISA.NET defined .NET events. VISA.NET defines events that are specific to a particular session interface and event type. These .NET events provide callback delegates, registration methods, and a notification mechanism that are specific, for the most part, to particular events. When using .NET events, registering for the event roughly corresponds to calling `viInstallHandler()` followed by `viEnableEvent()` with `VI_HNDLR` and unregistering the event roughly corresponds to calling `viDisableEvent()` followed by `viUninstallHandler()`. There are no equivalents to `viDiscardEvent()` and `viWaitOnEvent()` when using .NET events. A .NET event calls an event delegate (e.g. callback method) to handle the event.

VISA.NET uses standard .NET event mechanisms for registering and firing events. Each event delegate has an event arguments parameter (also called event args) to communicate information back and forth between VISA.NET and the calling program. Event delegates use either the VISA.NET default for event args (the `VisaEventArgs` class) or custom event args that derive from `VisaEventArgs`, depending on the event.

Events and event delegates are defined with the .NET `EventHandler<T>` delegate where T is the type of the event args.

VISA.NET can synchronize the execution of the callback functions that handle events and asynchronous I/O so that event handlers (for events) and callback functions (for asynchronous I/O) run in the caller's context. If callbacks are synchronized, VISA.NET captures the context when the calling program registers for the event or (for asynchronous I/O) BeginRead or BeginWrite is executed. It then uses the standard .NET mechanisms to ensure that events are fired, or callbacks are made in the caller's original context. Refer to section 8.3.2, *SynchronizeCallbacks*, for details.

CORRESPONDING VISA EVENTS

The following table shows the relationship between events in VISA and hardware events in VISA.NET.

VISA Event Name	VISA.NET EventType Used with Event Methods and EventArgs	VISA.NET .NET Events
Any vendor specific event code.	`Custom`	.NET event is vendor defined
`VI_EVENT_TRIG`	`Trigger`	`IGpibInterfaceSession.Trigger`
		`IVxiBackplaneSession.Trigger`
		`IVxiSession.Trigger`
`VI_EVENT_SERVICE_REQ`	`ServiceRequest`	`IGpibInterfaceSession.ServiceRequest`
		`IMessageBasedSession.ServiceRequest`
`VI_EVENT_CLEAR`	`Clear`	`IGpibInterfaceSession.Cleared`
`VI_EVENT_EXCEPTION`	No VISA.NET event	No VISA.NET event
`VI_EVENT_GPIB_CIC`	`GpibControllerInCharge`	`IGpibInterfaceSession.ControllerInCharge`
`VI_EVENT_GPIB_TALK`	`GpibTalk`	`IGpibInterfaceSession.Talk`
`VI_EVENT_GPIB_LISTEN`	`GpibListen`	`IGpibInterfaceSession.Listen`
`VI_EVENT_VXI_VME_SYSFAIL`	`VxiVmeSystemFailure`	`IVxiBackplaneSession.SystemFailure`
`VI_EVENT_VXI_VME_SYSRESET`	`VxiVmeSystemReset`	`IVxiBackplaneSession.SystemReset`
`VI_EVENT_VXI_SIGP`	`VxiSignalP`	`IVxiSession.SignalProcessor`
`VI_EVENT_VXI_VME_INTR`	`VxiVmeInterrupt`	`IVxiSession.Interrupt`
`VI_EVENT_PXI_INTR`	`PxiInterrupt`	`IPxiSession.Interrupt`
`VI_EVENT_TCPIP_CONNECT`	No VISA.NET event	No VISA.NET event
`VI_EVENT_USB_INTR`	`UsbInterrupt`	`IUsbSession.Interrupt`
`VI_ALL_ENABLED_EVENTS`	`AllEnabled`	Not meaningful, since the calling program has already registered for the "enabled" events.
`VI_EVENT_IO_COMPLETION`	Refer to the discussion of events in VISA.NET Asynchronous I/O.	

IMPLEMENTATION

OBSERVATION 7.1.1

VISA.NET does not support `VI_EVENT_EXCEPTION`. Exceptions are reported by a .NET exception in VISA.NET.

OBSERVATION 7.1.2

VISA.NET does not support `VI_EVENT_TCPIP_CONNECT`. This event type is only used with SERVANT sessions, which are not supported in VISA.NET.

OBSERVATION 7.1.3

VISA.NET I/O implementations should not assume an event handler will return in any timeframe. Event handlers may execute blocking waits before returning to the VISA.NET I/O component that fired the event. If a VISA.NET I/O resource component calls an event handler and the event handler blocks, the event handler will not return until the block completes.

OBSERVATION 7.1.4

Event handlers may affect the liveness of the VISA.NET I/O session making the calls. To prevent issues related to responsiveness, event handlers should make every effort to return in a timely manner.

RULE 7.1.2

VISA.NET implementations **SHALL NOT** kill threads which they did not start.

RECOMMENDATION 7.1.2

VISA.NET implementations should not hold synchronization objects that would prevent an event handler or callback routine from executing properly if the event handler or callback routine were to call back into VISA.NET.

RECOMMENDATION 7.1.3

If a VISA.NET I/O resource component calls an event handler or callback method which in turn throws an exception, VISA.NET should catch the exception without re-throwing it. This assumes that the customer has dealt with any exceptions, since the exception was generated from their code. Vendors need to document that users need to use try/catch in their event handlers or callbacks to handle exceptions.

OBSERVATION 7.1.5

As the effect of an exception leaving the context of an event handler or callback method is not deterministic, the event handler or callback method should make every effort to not allow this to happen.

7.2. .NET Event Handlers

VISA.NET defines the following event handlers. The following list shows the name of each standard event handler, the interface in which it is defined, and the event handler and delegate definition, including the event args class used with the event.

Message Based ServiceRequest in IMessageBasedSession

```
event EventHandler<VisaEventArgs> ServiceRequest;
```

GPIB Interface Cleared in IGpibInterfaceSession

```
event EventHandler<VisaEventArgs> Cleared;
```

GPIB Interface ControllerInCharge in IGpibInterfaceSession

```
event EventHandler<GpibControllerInChargeEventArgs> ControllerInCharge;
```

GPIB Interface Listen in IGpibInterfaceSession

```
event EventHandler<VisaEventArgs> Listen;
```

GPIB Interface ServiceRequest in IGpibInterfaceSession

```
event EventHandler<VisaEventArgs> ServiceRequest;
```

GPIB Interface Talk in IGpibInterfaceSession

```
event EventHandler<VisaEventArgs> Talk;
```

GPIB Interface Trigger in IGpibInterfaceSession

```
event EventHandler<VisaEventArgs> Trigger;
```

PXI Interrupt in IPxiSession

```
event EventHandler<PxiInterruptEventArgs> Interrupt;
```

USB Interrupt in IUsbSession

```
event EventHandler<UsbInterruptEventArgs> Interrupt;
```

VXI Backplane Trigger in IVxiBackplaneSession

```
event EventHandler<VxiTriggerEventArgs> Trigger;
```

VXI Backplane System Failure in IVxiBackplaneSession

```
event EventHandler<VisaEventArgs> SystemFailure;
```

VXI Backplane System Reset in IVxiBackplaneSession

```
event EventHandler<VisaEventArgs> SystemReset;
```

VXI Interrupt in IVxiSession

```
event EventHandler<VxiInterruptEventArgs> Interrupt;
```

VXI Signal Processor in `IVxiSession`

```
event EventHandler<VxiSignalProcessorEventArgs> SignalProcessor;
```

VXI Trigger in IVxiSession

```
event EventHandler<VxiTriggerEventArgs> Trigger;
```

7.3. VISA.NET Event Arguments

In .NET, every event handler has an event argument class that is used to communicate information between the routine that fires the event and the event handler. VISA.NET defines the following event argument classes.

- `VisaEventArgs`
- `GpibControllerInChargeEventArgs : VisaEventArgs`
- `PxiInterruptEventArgs : VisaEventArgs`
- `UsbInterruptEventArgs : VisaEventArgs`
- `VxiInterruptEventArgs : VisaEventArgs`
- `VxiSignalProcessorEventArgs : VisaEventArgs`
- `VxiTriggerEventArgs : VisaEventArgs`

VISA.NET also defines an interface that is used in conjuction with VISA.NET event arguments when the VISA.NET implementation delegates to an underlying VISA C implementation. This interface allows calling programs to retrieve native VISA C attribute values.

- `INativeVisaEventArgs`

7.3.1. VisaEventArgs Class

DESCRIPTION

The `VisaEventArgs` class communicates information about the event being fired. For events defined as part of this specification, the event is identified by a member of the `EventType` enumeration. For vendor specific events, each event is identified by a unique integer.

OBSERVATION 7.3.1

All of the other VISA.NET standard event argument classes defined in this specification derive from `VisaEventArgs`.

DEFINITION

```
public class VisaEventArgs : EventArgs
{
   public VisaEventArgs(EventType eventType) {…}
   public VisaEventArgs(Int32 customType) {…}
   public EventType EventType { get; private set; }
   public Int32 CustomEventType { get; private set; }
}
```

ARGUMENTS

Name	Description	Type
`eventType`	The VISA.NET standard event type.	`EventType`
`customType`	A value that uniquely identifies an implementation-specific event type.	`Int32`

PROPERTIES

Name	Description	Type
`EventType`	The VISA.NET standard event type.	`EventType`
`CustomType`	A value that uniquely identifies an implementation-specific event type.	`Int32`

CORRESPONDING VISA FEATURES

The `VisaEventArgs` class has COM properties that corresponds to an attribute defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`EventType`	`VI_ATTR_EVENT_TYPE`
`CustomEventType`	`VI_ATTR_EVENT_TYPE`

IMPLEMENTATION

`VisaEventArgs` is implemented in the VISA.NET standard components.

PERMISSION 7.3.1

Vendors may override the implementation of `VisaEventArgs`.

RULE 7.3.1

If `VisaEventArgs` is instantiated with a standard event type, the `CustomType` property **SHALL** return the value of the `EventType` enumeration member to which the constructor's `eventType` argument was set.

RULE 7.3.2

If `VisaEventArgs` is instantiated with a custom event type, the `EventType` property **SHALL** return `EventType.Custom`.

RULE 7.3.3

Vendor specific implementations of VISA.NET **SHALL NOT** define custom events with values that are identical to any of the values assigned to members of the `EventType` enumeration. Refer to VPP-4.3.2, Section 3.8, *Miscellaneous*, for legal ranges for vendor defined events.

7.3.2. GpibControllerInChargeEventArgs

DESCRIPTION

Provides additional data about a GPIB controller in charge (CIC) event. In particular, it indicates whether the controller is in charge.

DEFINITION

```
public class GpibControllerInChargeEventArgs : VisaEventArgs
{
   public GpibControllerInChargeEventArgs(Boolean isControllerInCharge) {…}
   public Boolean IsControllerInCharge { get; private set; }
}
```

ARGUMENTS

Name	Description	Type
`isControllerInCharge`	True if the controller for the GPIB interface is in charge.	`Boolean`

PROPERTIES

Name	Description	Type
`IsControllerInCharge`	True if the controller for the GPIB interface is in charge.	`Boolean`

CORRESPONDING VISA FEATURES

The `GpibControllerInChargeEventArgs` class has a property that corresponds to an attribute defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`IsControllerInCharge`	`VI_ATTR_GPIB_RECV_CIC_STATE`

7.3.3. PxiInterruptEventArgs

DESCRIPTION

Provides additional data about a PXI interrupt event. In particular, it includes the index of the interrupt sequence that detected the interrupt condition and the first register that was read in the successful interrupt detection sequence.

DEFINITION

```
public class PxiInterruptEventArgs : VisaEventArgs
{
   public PxiInterruptEventArgs(Int16 sequence, Int32 data) {…}

   public Int16 Sequence { get; private set; }
   public Int32 Data { get; private set; }
}
```

ARGUMENTS

Inputs	Description	Type
`sequence`	The index of the interrupt sequence that detected the interrupt condition.	`Int16`
`data`	The first register that was read in the successful interrupt detection sequence.	`Int32`

PROPERTIES

Inputs	Description	Type
`Sequence`	The index of the interrupt sequence that detected the interrupt condition.	`Int16`
`Data`	The first register that was read in the successful interrupt detection sequence.	`Int32`

CORRESPONDING VISA FEATURES

The `PxiInterruptEventArgs` class has properties that correspond to attributes defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`Sequence`	`VI_ATTR_PXI_RECV_INTR_SEQ`
`Data`	`VI_ATTR_PXI_RECV_INTR_DATA`

7.3.4. UsbInterruptEventArgs

DESCRIPTION

Provides additional data about a USB interrupt event. In particular, it includes the data that was being transferred over the USB bus, and indicates whether the data exceeded the maximum size expected from this device.

DEFINITION

```
public class UsbInterruptEventArgs : VisaEventArgs
{
   public UsbInterruptEventArgs(Boolean exceededMaximumSize, Byte[] data) {…}

   public Boolean ExceededMaximumSize { get; private set; }
   public Byte[] Data { get; private set; }
}
```

ARGUMENTS

Inputs	Description	Type
`exceededMaximumSize`	True if the data size exceeded the maximum size expected from this device.	`Boolean`
`data`	The data being transferred over the USB bus when the interrupt occurred, no longer than the expected maximum size. If no data was received, this argument **SHALL** be null.	`Byte[]`

PROPERTIES

Inputs	Description	Type
`ExceededMaximumSize`	True if the data size exceeded the maximum size expected from this device.	`Boolean`
`Data`	The data being transferred over the USB bus when the interrupt occurred, no longer than the expected maximum size. If no data was received, this property is null.	`Byte[]`

CORRESPONDING VISA FEATURES

The `UsbInterruptEventArgs` class has properties that correspond to attributes defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`Data (data array content)`	`VI_ATTR_USB_RECV_INTR_DATA`
`Data (data array size)`	`VI_ATTR_USB_RECV_INTR_SIZE`
`ExceededMaximumSize`	`VI_ATTR_STATUS`

7.3.5. VxiSignalProcessorEventArgs

DESCRIPTION

Provides additional data about a VXIbus signal or VXIbus interrupt event. In particular, it includes the status ID included with the interrupt.

DEFINITION

```
public class VxiSignalProcessorEventArgs : VisaEventArgs
{
   public VxiSignalProcessorEventArgs(Int32 statusId) {…}

   public Int32 StatusId { get; private set; }
}
```

ARGUMENTS

Inputs	Description	Type
`statusId`	The status ID of the VXIbus signal or VXIbus interrupt.	`Int32`

PROPERTIES

Inputs	Description	Type
`StatusId`	The status ID of the VXIbus signal or VXIbus interrupt.	`Int32`

CORRESPONDING VISA FEATURES

The `VxiSignalProcessorEventArgs` class has a property that corresponds to an attribute defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`StatusID`	`VI_ATTR_SIGP_STATUS_ID`

7.3.6. VxiTriggerEventArgs

DESCRIPTION

Provides additional data about a VXI trigger event. In particular, it includes the trigger line that caused the event.

DEFINITION

```
public class VxiTriggerEventArgs : VisaEventArgs
{
   public VxiTriggerEventArgs(TriggerLine triggerLine) {…}

   public TriggerLine TriggerLine { get; private set; }
}
```

ARGUMENTS

Inputs	Description	Type
`triggerLine`	The trigger line that caused the event.	`Ivi.Visa.TriggerLine`

PROPERTIES

Inputs	Description	Type
`TriggerLine`	The trigger line that caused the event.	`Ivi.Visa.TriggerLine`

CORRESPONDING VISA FEATURES

The `VxiTriggerEventArgs` class has a property that corresponds to an attribute defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`TriggerLine`	`VI_ATTR_RECV_TRIG_ID`

7.3.7. VxiInterruptEventArgs

DESCRIPTION

Provides additional data about a VXI interrupt event. In particular, it includes the interrupt level and the status ID included with the interrupt.

DEFINITION

```
public class VxiInterruptEventArgs : VisaEventArgs
{
   public VxiInterruptEventArgs(Int16 irqLevel, Int32 statusId) {…}

   public Int16 IrqLevel { get; private set; }
   public Int32 StatusId { get; private set; }
}
```

ARGUMENTS

Inputs	Description	Type
`irqLevel`	The interrupt level of the VXI VME interrupt.	`Int16`
`statusId`	The status ID of the VXI interrupt.	`Int32`

PROPERTIES

Inputs	Description	Type
`IrqLevel`	The interrupt level of the VXI VME interrupt.	`Int16`
`StatusId`	The status ID of the VXI interrupt.	`Int32`

CORRESPONDING VISA FEATURES

The `VxiInterruptEventArgs` class has properties that correspond to attributes defined in VISA. The following table shows property-attribute correspondence for each property.

Property Name	VISA Attribute Name
`IrqLevel`	`VI_ATTR_RECV_INTR_LEVEL`
`StatusID`	`VI_ATTR_INTR_STATUS_ID`

7.3.8. INativeVisaEventArgs Interface

DESCRIPTION

The `VisaEventArgs` class communicates the event being fired. For events defined as part of this specification, the event is identified by a member of the `EventType` enumeration. For vendor specific events, each event is identified by a unique integer.

DEFINITION

```
public interface INativeVisaEventArgs : IDisposable
{
   VisaEventArgs EventArgs { get; }

   Byte GetAttributeByte(NativeVisaAttribute attribute);
   Byte GetAttributeByte(Int32 attribute);

   Int16 GetAttributeInt16(NativeVisaAttribute attribute);
   Int16 GetAttributeInt16(Int32 attribute);

   Int32 GetAttributeInt32(NativeVisaAttribute attribute);
   Int32 GetAttributeInt32(Int32 attribute);

   Int64 GetAttributeInt64(NativeVisaAttribute attribute);
   Int64 GetAttributeInt64(Int32 attribute);

   Boolean GetAttributeBoolean(NativeVisaAttribute attribute);
   Boolean GetAttributeBoolean(Int32 attribute);

   String GetAttributeString(NativeVisaAttribute attribute);
   String GetAttributeString(Int32 attribute);
}
```

ARGUMENTS

Inputs	Description	Type
`attribute`	A constant that identifies a VISA standard attribute. The type of the method must match the type of the attribute.	`NativeVisaAttribute`
	A constant that identifies a VISA standard or vendor-defined attribute. The type of the method must match the type of the attribute.	`Int32`

PROPERTIES

Inputs	Description	Type
`EventArgs`	An event args reference for the native C event. This may reference an object that derives from EventArgs, if the event handler for the event specified by the EventArg.EventType or EventArg.CustomEventType properties uses a derived class.	`EventArgs`

CORRESPONDING VISA FEATURES

The INative`VisaEventArgs` interface has several methods that correspond to VISA functions. The following table shows method-function correspondence for each method.

Method Name	**VISA Attribute Name**
`GetAttributeByte`	`viGetAttribute`
`GetAttributeInt16`	`viGetAttribute`
`GetAttributeInt32`	`viGetAttribute`
`GetAttributeInt64`	`viGetAttribute`
`GetAttributeBoolean`	`viGetAttribute`
`GetAttributeString`	`viGetAttribute`

IMPLEMENTATION

RULE 7.3.4

Implementors **SHALL** call `viClose` on the underlying native VISA event object only when the user disposes the object that implements `INativeVisaEventArgs`.

RECOMMENDATION 7.3.1

For vendor-specific attributes, vendors should give guidance on which GetAttribute method to use based on the VISA C type of the attribute being retrieved.

7.4. Vendor Defined Events

Implementors may create vendor specific events.

IMPLEMENTATION

RULE 7.4.1

All vendor specific VISA.NET event argument classes **SHALL** either be `VisaEventArgs` or be derived from `VisaEventArgs` directly or indirectly.

PERMISSION 7.4.1

Vendor specific implementations may define event handlers using any of the ways allowed by .NET. They are not required to use the `EventHandler<T>` delegate.

7.5. Event Methods

As mentioned above, VISA.NET event methods include `EnableEvent()`, `DisableEvent()`, `DiscardEvent()`, and `WaitOnEvent()`, which exactly correspond to the C functions `viEnableEvent()` (called with an event mechanism of `VI_QUEUE`), `viDisableEvent()`, `viDiscardEvent()`, and `viWaitOnEvent()`. These methods are part of `IVisaSession`, and so are included with every type of VISA.NET session. Refer to *VPP-3.4: The VISA Library*, section 3.7, *Event Services*, and particularly section 3.7.1, *Event Handling and Processing*, section 3.7.31, *viEnableEvent*, section 3.7.3.2, *viDisableEvent*, section 3.7.3.3, *viDiscardEvent*, and section 3.7.3.4, *viWaitOnEvent*, for details.

For the definition of the event methods in VISA.NET, refer to section 8.3, *IVisaSession* Interface

When using event methods, the event mechanism is always a queue (`VI_QUEUE`).

The return type of the `WaitOnEvent` method is defined as `VisaEventArgs`, which gives the client information about the event from the server. This is the same `VisaEventArgs` that is the base class for all of the event args defined for VISA.NET's .NET events. The data returned by the `WaitOnEvent` method is either `VisaEventArgs` or derived from `VisaEventArgs`. The exact return type can be inferred from the value of `VisaEventArgs.EventType` and `VisaEventArgs.CustomEventType`.

Section 8: VISA.NET Sessions

In general terms, a session represents a connection to a unique hardware *resource* (instrument, interface, backplane, etc.) using a particular kind of *I/O protocol*. In VISA.NET, a *session* is an instance of a VISA.NET class that is used to communicate with a specific resource. All of the I/O in VISA.NET happens in sessions.

8.1. Session Overview

VISA.NET supports a variety of different types of sessions, which vary by the I/O protocol and the resource class of the session interface. Each connected resource is identified by a *resource descriptor* that uniquely identifies the resource. A *resource manager* is capable of accepting a resource descriptor and returning a session that is ready to use for I/O.

All VISA.NET sessions have some capabilities in common. There are also two broad subcategories of sessions, *message-based sessions* and *register based sessions*. Each of these subcategories also has some common capabilities. Finally, each session type has capabilities specific to that type. All of these capabilities are represented in a hierarchy of VISA.NET *session interfaces*.

Between the the resource manager and the session interfaces, VISA.NET presents a full set of capabilities related to the session lifecycle, locking, event handling, and resource and I/O specific functionality.

8.1.1. Resources and Resource Descriptors

Resources are categorized into *resource classes*. The most common resource class is a straightforward connection to an instrument. Refer to VPP-4.3, Section (TODO) for a list of resource classes and their acronyms. Note that the SERVANT resource class is not supported in VISA.NET.

Each resource on a system is identified by a unique resource ID called a *resource descriptor* or *resource name*. Resource names begin with the hardware interface type and number followed by "::", and end with "::" followed by a resource class. The information between the first "::" and the last uniquely identifies the hardware within the hardware interface type and the resource class. Refer to VPP-4.3, Section (TODO) for a more complete description of resource names.

In addition to the hardware interfaces that are supported by VISA.NET, vendors may add vendor specific hardware interfaces and corresponding session types that conform to the requirements for the common elements that all of the resource types share. In such cases, the vendor specific session interface must ultimately derive from `IVisaSession` and may derive from `IMessageBasedSession` and/or `IRegisterBasedSession`.

8.1.2. Resources Managers

Each VISA.NET session class must include a constructor that creates a session and initializes a VISA.NET I/O Resource. However, the recommended way to create the session is to use a VISA.NET *resource manager*. There are two types of resource manager, vendor specific resource managers and the VISA.NET Shared Components Global Resource Manager (GRM). Refer to Section 17: Resource Manager Classes for a detailed description of these classes.

8.1.3. Session Interfaces

All session interfaces include some elements that are common to all sessions. These elements are defined in the interface `IVisaSession` All VISA.NET session interfaces ultimately derive from `IVisaSession`, and so include this common functionality. For VISA.NET session implementations that delegate to an underlying VISA C implementation, vendors may also choose to implement `INativeVisaSession`, which exposes methods that enable clients to access vendor-specific C attributes and events.

Nearly all session interfaces are either message-based or register-based. All message-based elements are defined in the interface `IMessageBasedSession`. All VISA.NET message-based session interfaces derive from `IMessageBasedSession`, and so include this common functionality.

Message-based sessions provide access to two different ways of performing message-based I/O. The first is *raw I/O*, which leaves formatting and parsing tasks (including format buffering) to the calling program. Raw I/O supports asynchronous I/O. The second is *formatted I/O*, which is capable of some very complex formatting tasks, and supports a wide variety of formatting options. While it migh be tempting to think of formatted I/O for convenience and raw I/O for performance, in fact formatted I/O is highly optimized for performance.

Likewise, all register-based elements are defined in the interface `IRegisterBasedSession`. All VISA.NET register-based session interfaces derive from `IRegisterBasedSession`, and so include this common functionality.

8.1.4. Locking

Calling programs can open multiple sessions to a VISA.NET I/O resource simultaneously. Applications can access the VISA.NET I/O resource through these different sessions concurrently. To avoid conflicting behavior, a session accessing a VISA.NET I/O resource might want to restrict other sessions from accessing that resource. VISA defines a locking mechanism to restrict how multiple session access the same resource. These mechanisms are supported by the resource manager `Open()` method and by the `LockResource()` and `UnlockResource()` methods in the `IVisaSession` interface.

8.2. Session Interfaces

VISA.NET defines the following base interfaces for sessions. Derived interfaces are shown to make the hierarchy of interfaces clear. Note that `IMessageBasedSession` contains references to two interfaces that extend message based functionality: `IMessageBasedRawIO` and `IMessageBasedFormattedIO`.

- `IVisaSession`
- `INativeVisaSession : IVisaSession`
- `IMessageBasedSession : IVisaSession`
    - `IMessageBasedRawIO`
    - `IMessageBasedFormattedIO`
- `IRegisterBasedSession : IVisaSession`

`IVisaSession`, `INativeVisaSession`, `IMessageBasedSession`, and `IRegisterBasedSession` are not implemented directly for the interface types covered by this specification. The following session interfaces may be implemented directly by a VISA.NET implementation. One level of inheritance is shown to make the hierarchy of interfaces clear.

- `IGpibInterfaceSession : IVisaSession`
- `IPxiBackPlaneSession : IVisaSession`
- `IVxiBackplaneSession : IVisaSession`
- `IGpibSession : IMessageBasedSession`
- `ISerialSession : IMessageBasedSession`
- `ITcpipSession : IMessageBasedSession`
- `ITcpipSocketSession : IMessageBasedSession`
- `IUsbSession : IMessageBasedSession`
- `IVxiSession : IMessageBasedSession, IRegisterBasedSession`
- `IPxiSession : IRegisterBasedSession`
- `IPxiMemorySession : IRegisterBasedSession`
- `IVxiMemorySession : IRegisterBasedSession`

IMPLEMENTATION

RULE 8.2.1

A VISA.NET implementation **SHALL** implement at least one session interface. The interface may be one of the interfaces in the list of directly implementable interfaces above, or a vendor specific session class or interface that derives from `IVisaSession`, `INativeVisaSession`, `IMessageBasedSession`, or `IRegisterBasedSession`.

8.3. IVisaSession Interface

DESCRIPTION

This section summarizes `IVisaSession`, the interface from which every VISA.NET session must derive. For the interfaces defined in this specification, `IVisaSession` is never implemented directly. Rather, one of the specializations of `IVisaSession` is implemented. `IVisaSession` provides common functionality for all of the specializations.

DEFINITION

```
public interface IVisaSession : IDisposable
{
   Int32 TimeoutMilliseconds { get; set; }
   String ResourceName { get; }
   String HardwareInterfaceName { get; }
   HardwareInterfaceType HardwareInterfaceType { get; }
   Int16 HardwareInterfaceNumber { get; }
   String ResourceClass { get; }
   String ResourceManufacturerName { get; }
   Int16 ResourceManufacturerId { get; }
   Version ResourceImplementationVersion { get; }
   Version ResourceSpecificationVersion { get; }
   ResourceLockState ResourceLockState { get; }

   void LockResource();
   void LockResource(TimeSpan timeout);
   void LockResource(Int32 timeoutMilliseconds);
   string LockResource(TimeSpan timeout, String sharedKey);
   string LockResource(Int32 timeoutMilliseconds, String sharedKey);
   void UnlockResource();

   Int32 EventQueueCapacity { get; set; }
   Boolean SynchronizeCallbacks { get; set; }

   void EnableEvent(EventType eventType);
   void DisableEvent(EventType eventType);
   void DiscardEvent(EventType eventType);
   VisaEventArgs WaitOnEvent(EventType eventType);
   VisaEventArgs WaitOnEvent(EventType eventType,
                             out EventQueueStatus status);
   VisaEventArgs WaitOnEvent(EventType eventType, Int32 timeoutMilliseconds);
   VisaEventArgs WaitOnEvent(EventType eventType, TimeSpan timeout);
   VisaEventArgs WaitOnEvent(EventType eventType, Int32 timeoutMilliseconds,
                             out EventQueueStatus status);
   VisaEventArgs WaitOnEvent(EventType eventType, TimeSpan timeout,
                             out EventQueueStatus status);
}
```

CORRESPONDING VISA FEATURES

The `IVisaSession` Interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IVisaSession`.

Property Name	VISA Attribute Name
EventQueueCapacity	VI_ATTR_MAX_QUEUE_LENGTH
HardwareInterfaceName	VI_ATTR_INTF_INST_NAME
HardwareInterfaceType	VI_ATTR_INTF_TYPE
HardwareInterfaceNumber	VI_ATTR_INTF_NUM
ResourceClass	VI_ATTR_RSRC_CLASS
ResourceImplementationVersion	VI_ATTR_RSRC_IMPL_VERSION
ResourceLockState	VI_ATTR_RSRC_LOCK_STATE
ResourceManufacturerName	VI_ATTR_RSRC_MANF_NAME
ResourceManufacturerID	VI_ATTR_RSRC_MANF_ID
ResourceName	VI_ATTR_RSRC_NAME
ResourceSpecificationVersion	VI_ATTR_RSRC_SPEC_VERSION
SynchronizeCallbacks	N/A
TimeoutMilliseconds	VI_ATTR_TMO_VALUE

The `IVisaSession` Interface has several methods that map to VISA functions. The following table shows VISA equivalence for `IVisaSession` methods.

Method Name	VISA Method Name
DisableEvent	viDisableEvent
DiscardEvents	viDiscardEvents
EnableEvent	viEnableEvent
LockResource	viLock
UnlockResource	viUnlock
WaitOnEvent	viWaitOnEvent
IDisposable.Dispose	viClose

OBSERVATION 8.3.1

There is not an exact mapping between `IVisaSession` and the VISA Resource Template. Because the properties `HardwareInterfaceNumber`, `TimeoutMilliseconds`, `HardwareInterfaceName`, and `HardwareInterfaceType` are used by all resource types, they have been placed to `IVisaSession` to maximize polymorphism.

OBSERVATION 8.3.2

There is not an exact mapping between `LockResource()` and VISA's `viLock()`, but there is a strong correspondence between the overloads of `LockResource()` and `viLock()` The overloads of `LockResource()` that return `void` obtain an exclusive lock. The overloads that return `String` obtain a shared lock with the specified key. Regardless of whether an exclusive or shared lock is being requested, it is possible to specify a timeout in milliseconds or as a time span.

OBSERVATION 8.3.3

The `LockResource()` and `UnlockResource()` methods do not support VISA's `viLock()` alternate success codes.

OBSERVATION 8.3.4

The `EnableEvent(),DisableEvent(), and DiscardEvent()` methods are the same as the corresponding C functions with `mechanism` set to `VI_QUEUE`.

OBSERVATION 8.3.5

There is not an exact mapping between `WaitOnEvent()` and VISA's `viWaitOnEvent()`, but there is a strong correspondence between them. The `WaitOnEvent()` overloads do not have an out `EventType` argument.

OBSERVATION 8.3.6

In the VISA C API, `viWaitOnEvent()` returns a positive value to indicate a warning or to provide additional information about a successful call. The `status` argument to `WaitOnEvent ()` is used to indicate the equivalent information. Note that it is an `out` argument.

OBSERVATION 8.3.7

VISA.NET does not contain any method or property corresponding to VISA's `VI_ATTR_RSRC_RM_SESSION` or `VI_ATTR_USER_DATA` attributes.

OBSERVATION 8.3.8

The VISA.NET defined special value for an infinite timeout is `VisaConstants.InfiniteTimeout`. The value is -1, which is the same 32-bit value as `VI_TMO_INFINITE`.

OBSERVATION 8.3.9

Negative timeout values other than -1 may or may not be recognized as unsigned integer values, based on the vendor implementation.

OBSERVATION 8.3.10

Timeout parameters whose type is specified as TimeSpan support a timespan in milliseconds that matches the range of the Int32 timeout values at a minimum, but may support longer timeout values based on the vendor implementation. For these parameters, the VISA.NET special value for an infinite timeout is TimeSpan.MaxValue.

IMPLEMENTATION

RULE 8.3.2

VISA.NET I/O session classes SHALL implement `IVisaSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 8.3.3

VISA.NET I/O session classes **SHALL** follow the semantics defined in section 3.2 of VPP 4.3 with the exceptions noted above.

RULE 8.3.4

Every VISA.NET I/O session class **SHALL** derive from `IVisaSession`, or from an interface that derives from `IVisaSession`.

RULE 8.3.5

The `Dispose`() method **SHALL** cause the resource to clean itself up, and **SHALL** destroy the .NET object.

RULE 8.3.6

For a VISA.NET implementation that calls an underlying VISA C implementation, the `Dispose()` method **SHALL** call `viClose()`.

RULE 8.3.7

The value of the attribute `ResourceSpecificationVersion` **SHALL** be the following:

- MajorVersion **SHALL** be the major version of this specification, as shown on the title page.

- MinorVersion **SHALL** be the minor version of this specification, as shown on the title page.
- BuildNumber and Revision **SHALL** be 0.

For example, the value of `ResourceSpecificationVersion` for version 5.4 of this specification would be 5.4.0.0.

RULE 8.3.8

The value of `ResourceSpecificationVersion` for a particular resource SHALL be the oldest specification version of all of the VISA-compliant binaries that are invoked in the implementation of the resource instance, including the VISA.NET specification version of the VISA.NET assembly used.

OBSERVATION 8.3.11

The above rule implies that the `ResourceSpecificationVersion` for a particular resource identifies the VISA functionality which the resource provides. For example, if a VISA.NET resource is based on version 6.0 of the VISA.NET specification, but invokes a VISA C implementation based on version 5.4 of the VISA C specification, `ResourceSpecificationVersion` would report version5.4.

OBSERVATION 8.3.12

Session classes across multiple implementations of VISA.NET, whether from a single vendor or multiple vendors, may have different values for `ResourceSpecificationVersion` depending on the vendor and implementation version.

RULE 8.3.9

If a session class implements a vendor specific hardware interface type, that class **SHALL** return `HardwareInterfaceType=HardwareInterfaceType.Custom`.

RULE 8.3.10

For session classes that implement VISA.NET defined interface types, the `HarwareInterfaceType` property **SHALL** return the corresponding `HardwareInterfaceType` value and the `HardwareInterfaceName` property **SHALL** include `"ASRL"`, `"GPIB"`, `"GPIB-VXI"`, `"PXI"`, `"TCPIP"`, `"USB"`, or `"VXI"` followed by the interface number.

RULE 8.3.11

For session classes that implement vendor specific interface types, the `HarwareInterfaceType` property **SHALL** return `HardwareInterfaceType.Custom` and the `HardwareInterfaceName` property **SHALL** include a string that identifies the interface type (not `"ASRL"`, `"GPIB"`, `"GPIB-VXI"`, `"PXI"`, `"TCPIP"`, `"USB"`, or `"VXI")` followed by an interface number.

8.3.2. SynchronizeCallbacks

DESCRIPTION

Specifies whether callbacks must be performed in a specific synchronization context. If `false`, the implementation is allowed to execute callbacks in any context.

This property applies to both I/O callbacks and events, but the point in time at which the synchronization context is captured is different. For events, context is captured at event registration, for each event type and delegate, regardless of the current state of this property. When an event is raised, the implementation uses this property to determine the context in which to invoke the delegate. For asynchronous I/O, context is captured at the begin operation (e.g. `BeginRead`, `BeginWrite`) if this property is `true`.

The default value is `true`.

DEFINITION

```
Boolean SynchronizeCallbacks { get; set; }
```

8.4. INativeVisaSession Interface

DESCRIPTION

This section summarizes `INativeVisaSession`, which allows access to vendor specific C attributes and events. For the interfaces defined in this specification, `INativeVisaSession` is never implemented directly. Rather, one of the specializations of `IVisaSession` also implements `INativeVisaSession` if the implementation delegates to VISA C. `INativeVisaSession` provides common functionality for all of the specializations.

DEFINITION

```
public interface INativeVisaSession : IVisaSession
{
   Int32 Handle { get; }

   void EnableEvent(Int32 eventType);
   void DisableEvent(Int32 eventType);
   void DiscardEvents(Int32 eventType);

   INativeVisaEventArgs WaitOnEvent(Int32 eventType);
   INativeVisaEventArgs WaitOnEvent(Int32 eventType,
                                    out EventQueueStatus status);
   INativeVisaEventArgs WaitOnEvent(Int32 eventType,
                                    Int32 timeoutMilliseconds);
   INativeVisaEventArgs WaitOnEvent(Int32 eventType, TimeSpan timeout);
   INativeVisaEventArgs WaitOnEvent(Int32 eventType, Int32 timeoutMilliseconds,
                                    out EventQueueStatus status);
   INativeVisaEventArgs WaitOnEvent(Int32 eventType, TimeSpan timeout,
                                    out EventQueueStatus status);

   Byte GetAttributeByte(NativeVisaAttribute attribute);
   Byte GetAttributeByte(Int32 attribute);

   Int16 GetAttributeInt16(NativeVisaAttribute attribute);
   Int16 GetAttributeInt16(Int32 attribute);

   Int32 GetAttributeInt32(NativeVisaAttribute attribute);
   Int32 GetAttributeInt32(Int32 attribute);

   Int64 GetAttributeInt64(NativeVisaAttribute attribute);
   Int64 GetAttributeInt64(Int32 attribute);

   Boolean GetAttributeBoolean(NativeVisaAttribute attribute);
   Boolean GetAttributeBoolean(Int32 attribute);

   String GetAttributeString(NativeVisaAttribute attribute);
   String GetAttributeString(Int32 attribute);

   void SetAttributeByte(NativeVisaAttribute attribute, Byte value);
   void SetAttributeByte(Int32 attribute, Byte value);
```

```
    void SetAttributeInt16(NativeVisaAttribute attribute, Int16 value);
    void SetAttributeInt16(Int32 attribute, Int16 value);

    void SetAttributeInt32(NativeVisaAttribute attribute, Int32 value);
    void SetAttributeInt32(Int32 attribute, Int32 value);

    void SetAttributeInt64(NativeVisaAttribute attribute, Int64 value);
    void SetAttributeInt64(Int32 attribute, Int64 value);

    void SetAttributeBoolean(NativeVisaAttribute attribute, Boolean value);
    void SetAttributeBoolean(Int32 attribute, Boolean value);

    void SetAttributeString(NativeVisaAttribute attribute, String value);
    void SetAttributeString(Int32 attribute, String value);
}
```

CORRESPONDING VISA FEATURES

The `INativeVisaSession` Interface has a .NET property that corresponds to the `vi` parameter defined in VISA. The following table shows the correspondence for `INativeVisaSession`.

Property Name	**VISA Attribute Name**
`Handle`	`vi` parameter returned by `viOpen()`

The `INativeVisaSession` Interface has several methods that map to VISA functions. The following table shows VISA equivalence for `INativeVisaSession` methods. Note that the VISA functions are not type specific, whereas the VISA.NET methods are.

Method Name	**VISA Function Name**
`EnableEvent`	`viEnableEvent`
`DisableEvent`	`viDisableEvent`
`DiscardEvents`	`viDiscardEvents`
`WaitOnEvent`	`viWaitOnEvent`
`GetAttribute<Type>`	`viGetAttribute`
`SetAttribute<Type>`	`viSetAttribute`

IMPLEMENTATION

OBSERVATION 8.4.1

A VISA.NET implementation is not required to implement `INativeVisaSession` even if the implementation delegates to an underlying VISA C I/O.

PERMISSION 8.4.1

A VISA.NET implementation may implement `INativeVisaSession` either implicitly or explicitly.

Section 9: Message Based Session Interfaces

Message based resources support basic stream I/O to instruments. While there are some special features that support 488.2, other basic message-based resources are supported. See VPP-4.3 section 5.1 for more information about these resources. The functionality of INSTR resources is broken up into several interfaces in VISA.NET I/O. Users can write code that polymorphically acts on any INSTR resource type by using only these resources and the Init string to create, instantiate, and use instruments.

9.1. IMessageBasedSession Interface

DESCRIPTION

This section summarizes `IMessageBasedSession`, the interface from which every VISA.NET message-based session must derive. Message based session classes defined in this specification implement interfaces that derive from `IMessageBasedSession`. `IMessageBasedSession` provides common functionality for all of the derived interfaces.

`IMessageBasedSession` includes a few basic message-based properties and methods, but the bulk of message based I/O is handled by two other interfaces, `IMessageBasedRawIO` and `IMessageBasedFormattedIO`. `IMessageBasedSession` contains properties that refer to these interfaces.

`IMessageBasedRawIO` allows calling programs to send string or byte array data to the instrument without any formatting or transformation. `IMessageBasedRawIO` may be synchronous or ansynchronous.

`IMessageBasedFormattedIO` formats data before sending it to the instrument. This means that calling programs can represent data in a variety of familiar numeric and enumerated types that are appropriate to the program and let VISA.NET do the work of formatting the data for the instrument. `IMessageBasedFormattedIO` operations are always synchronous. VISA.NET formatted I/O is optimized for IEEE-488, but works with many other message-based protocols as well.

DEFINITION

```
public interface IMessageBasedSession : IVisaSession
{
   event EventHandler<VisaEventArgs> ServiceRequest;

   IOProtocol IOProtocol { get; set; }
   Boolean SendEndEnabled { get; set; }
   Byte TerminationCharacter { get; set; }
   Boolean TerminationCharacterEnabled { get; set; }

   void AssertTrigger();
   void Clear();
   StatusByteFlags ReadStatusByte();

   IMessageBasedFormattedIO FormattedIO { get; }
   IMessageBasedRawIO RawIO { get; }
}
```

INTERFACE REFERENCES

The `IMessageBasedSession` interface has two properties that return references to other VISA.NET interfaces. The following table shows these properties and the interfaces to which they refer.

Property Name	Interface

FormattedIO	IMessageBasedFormattedIO
RawIO	IMessageBasedRawIO

CORRESPONDING VISA FEATURES

The IMessageBasedSession interface has several properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for IMessageBasedSession.[1]

Property Name	**VISA Attribute Name**
IOProtocol	VI_ATTR_IO_PROT
SendEndEnabled	VI_ATTR_SEND_END_EN
TerminationCharacter	VI_ATTR_TERMCHAR
TerminationCharacterEnabled	VI_ATTR_TERMCHAR_EN

The IMessageBasedSession interface has several methods that correspond to functions defined in VISA. The following table shows method-function correspondence for IMessageBasedSession.

Method Name	**VISA Function Name**
AssertTrigger	viAssertTrigger
Clear	viClear
ReadStatusByte	viReadSTB

The IMessageBasedSession interface has one .NET event that corresponds to an event defined in VISA. The following table shows correspondence for IMessageBasedSession.

Event Name	**VISA Function Name**
ServiceRequest	VI_EVENT_SERVICE_REQ

The IMessageBasedSession interface has one .NET event, ServiceRequest, that corresponds to functionality defined in VISA. There are some message based session types (for example, TCPIP SOCKET) that do not support service request events. For those session types, attempts to register a handler should fail with an exception.

IMPLEMENTATION

RULE 9.1.1

Message based VISA.NET I/O session classes **SHALL** implement IMessageBasedSession interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except where specified otherwise in this specification.

OBSERVATION 9.1.1

All VISA.NET I/O session classes that implement the GPIB, TCPIP, VXI, USB, GPIB-VXI, and ASRL INSTR resources derive from IMessageBasedSession indirectly.

RULE 9.1.2

When AssertTrigger is implemented by calling an underlying VISA, the underlying call to viAssertTrigger uses a protocol of VI_TRIG_PROT_DEFAULT.

If the FormattedIO member is not null, then the implementation of the Clear() method **SHALL** invoke FormattedIO.DiscardBuffers().

[1] The VISA attribute VI_ATTR_SUPPRESS_END_EN is intentionally not represented in the VISA.NET API as a property because it was intended to support old instruments that are not 488.2 compliant. It may be accessed using the INativeVisaSession interface.

9.2. IMessageBasedRawIO

DESCRIPTION

This section summarizes `IMessageBasedRawIO`. Note that `IMessageBasedRawIO` allows calling programs to send string or byte array data to the instrument without any formatting or parsing. This is contrasted to formatted I/O, which can format and parse a variety of data types.

`IMessageBasedRawIO` supports both synchronous and ansynchronous I/O.

`IMessageBasedSession` provides a property that returns a reference to `IMessageBasedRawIO`. This property is the only specified way to access the `IMessageBasedRawIO` interface from a message-based session.

DEFINITION

The `IMessageBasedRawIO` interface declaration is shown below. The body of the interface is documented in the sections that document individual properties and methods.

```
public interface IMessageBasedRawIO
```

CORRESPONDING VISA FEATURES

The `IMessageBasedRawIO` interface has several .NET methods that start asynchronous operations. These methods return a reference to `IVisaAsyncResult`, which returns information about the asynchronous operation implemented by the method. `IVisaAsyncResult` is described in detail below.

The `IMessageBasedRawIO` interface's methods that perform I/O correspond to functions defined in VISA. The following table shows method-function correspondence for `IMessageBasedRawIO`. Note that in most cases these methods are not equivalent to the functions due to slight differences in behavior between VISA C and VISA.NET. The methods that perform asynchronous operations are significantly different from VISA C. For this reason, all of the methods in `IMessageBasedRawIO` are described in detail below.

Method Name	**VISA Function Name**
`Write`	`viWrite`
`Read`	`viRead`
`ReadString`	`viRead`
`BeginWrite`	`viWriteAsync`
`EndWrite`	`viWaitOnEvent` (w/ IO Completion event)
`BeginRead`	`viReadAsync`
`EndRead`	`viWaitOnEvent` (w/ IO Completion event)
`AbortAsyncOperation`	`viTerminate`
`ReadAsync`	`viReadAsync, viWaitOnEvent, viTerminate`
`WriteAsync`	`viWriteAsync, viWaitOnEvent, viTerminate`

9.2.1. Synchronous I/O

The raw I/O synchronous methods perform the requested I/O and return only after the I/O operation is complete.

9.2.1.1. **Read**

DESCRIPTION

All overloads of the `Read` method read bytes from the device and return them as an array of bytes.

The overloads of the `Read` method that return an array of bytes allocate the array themselves. For other overloads, the calling program must allocate the array before making the call, and the array must contain at least `index + count` elements if `index` and `count` are used.

Bytes are returned exactly as they are read from the device, in exactly the same order.

Reading continues until one of the following conditions is met:

- An END indicator is read from the data coming from the device. This will only happen if END is supported by the protocol being used, and is enabled.
- A termination character is read in the data coming from the device, and `IMessageBasedSession.TerminationCharacterEnabled` is `true`. In this case, the termination character is included in the data buffer.
- Exactly `count` bytes have been read from the device, if the overload includes the `count` argument.
- The amount of time spent reading (or trying to read) data from the device exceeds `IVisaSession.TimeoutMilliseconds`, in which case an exception is thrown.

DEFINITION

```
Byte[] Read();
Byte[] Read(Int64 count);
Byte[] Read(Int64 count, out ReadStatus readStatus);

void Read(Byte[] buffer, Int64 index, Int64 count,
          out Int64 actualCount, out ReadStatus readStatus);
unsafe void Read(Byte* buffer, Int64 index, Int64 count,
          out Int64 actualCount, out ReadStatus readStatus);

#if NET6_0_OR_GREATER
IVisaReadResult Read(Span<Byte> buffer);
#endif
```

ARGUMENTS

Name	Description	Type
`count`	The maximum number of bytes to be returned from the device. The default is to read until an end condition is received.	`Int64`
`actualCount`	The actual count of bytes stored in the buffer parameter during the read operation.	`Int64`
`index`	In the array, the index where the method places the first byte returned from the device. The default is 0.	`Int64`

`buffer`	An array of bytes allocated by the calling program, into which bytes returned by the device are placed.
	A reference to the array
	A span of bytes.
		`Byte[]`
		`Byte*`
		`Span<Byte>`

`readStatus`	Indicates how the read terminated. If an END was received, `ReadStatus.EndReceived` is returned. Otherwise, if a termination character was received and `TerminationCharacterEnabled` is true, `ReadStatus.TerminationCharacterEncountered` is returned. Otherwise `ReadStatus.MaximumCountReached` is returned.	`ReadStatus`

RETURN VALUES

Name	Description	Type
`return value`	An array of bytes allocated by the method, into which bytes returned by the device are placed. The size of the array returned is the number of bytes actually read.	`Byte[]`
	The number of bytes that were read and placed into the `data` array.	`Int64`
	An interface reference that provides both the number of bytes that were read and placed into the `data` array and the read status of the operation.	`IVisaReadResult`

EXCEPTIONS

This method uses the `Ivi.Visa.IoTimeoutException` to report a timeout.

IMPLEMENTATION

OBSERVATION 9.2.1

`Count` and `index` parameters are not needed for the overload that uses Span<Byte> for the buffer. When the client creates the span object to pass to the method, it can create the span with the correct count and at the correct index if needed.

9.2.1.2. **ReadString**

DESCRIPTION

All overloads of the `ReadString` method reads characters from the device, converts them to a zero-extended UNICODE string, and returns the string.

Characters are returned in exactly the same order as they are read from the device.

Reading continues until one of the following conditions is met:

- An END indicator is read from the data coming from the device. This will only happen if END is supported by the protocol being used.
- A termination character is read in the data coming from the device, and `IMessageBasedSession.TerminationCharacterEnabled` is `true`. In this case, the termination character is included in the data.
- Exactly `count` characters have been read from the device, if the overload includes the `count` argument.
- The amount of time spent reading (or trying to read) data from the device exceeds `IVisaSession.TimeoutMilliseconds`, in which case an exception is thrown.

DEFINITION

```
String ReadString();
String ReadString(Int64 count);
String ReadString(Int64 count, out ReadStatus readStatus);
```

ARGUMENTS

Name	Description	Type
`count`	The maximum number of bytes to be returned from the device.	`Int64`
`readStatus`	Indicates how the read terminated. If an END was received, `ReadStatus.EndReceived` is returned. Otherwise, if a termination character was received and `TerminationCharacterEnabled` is true, `ReadStatus.TerminationCharacterEncountered` is returned. Otherwise `ReadStatus.MaximumCountReached` is returned.	`ReadStatus`

EXCEPTIONS

This method uses the `Ivi.Visa.IoTimeoutException` to report a timeout.

9.2.1.3. **Write**

DESCRIPTION

Overloads of the `Write` method that include a buffer argument of type `Byte[]`, `Byte*`, or `Span<Byte>` send the bytes to the device exactly as they appear in the array.

Overloads of the `Write` method that include a buffer argument of type `String` convert the string from UNICODE to 8-bit ASCII before sending it to the device. If the string contains a character that cannot be converted to an 8-bit ASCII character, the method throws an exception that identifies the invalid character.

Characters are written in exactly the same order as they occur in the array or string.

Writing continues until one of the following conditions is met:

- Exactly `count` characters have been written to the device, if the overload includes the `count` argument.
- The entire buffer has been written to the device, if the overload does not include the `count` argument.
- The amount of time spent writing (or trying to write) data to the device exceeds `IVisaSession.TimeoutMilliseconds`, in which case an exception is thrown.

An END is signaled with the last byte if `SendEndEnabled` is `true`.

Termination characters must be explicitly sent when writing to a device. The `Write` method does not send a termination character to the device that is not included in the buffer argument for all session types except those that define a `WriteTermination` property.

RULE 9.2.1
If the Write method is called with the parameter `count` smaller than the size of the array passed in, only the first `count` bytes **SHALL** be written to the instrument resource.

DEFINITION

```
void Write(Byte[] buffer);
void Write(Byte[] buffer, Int64 index, Int64 count);
void Write(String buffer);
void Write(String buffer, Int64 index, Int64 count);
unsafe void Write(Byte* buffer, Int64 index, Int64 count);

#if NET6_0_OR_GREATER
void Write(ReadOnlySpan<Byte> buffer);
#endif
```

ARGUMENTS

Name	Description	Type
`count`	The maximum number of bytes to be sent to the device.	`Int64`
`index`	In the array or string, the index of the first byte or character to be sent to the device.	`Int64`

`buffer`	The array or string to be sent to the device.
	
	A reference to the array.
	A span of bytes
		`Byte[]`
		`String`
		`Byte*`
		`ReadOnlySpan<Byte>`

EXCEPTIONS

This method uses the `Ivi.Visa.IoTimeoutException` to report a timeout.

IMPLEMENTATION

OBSERVATION 9.2.2

`Count` and `index` parameters are not needed for the overload that uses Span<Byte> for the buffer. When the client creates the span object to pass to the method, it can create the span with the correct count and at the correct index if needed.

9.2.1.4. **IVisaReadResult**

DESCRIPTION

After a Read method has completed, it is useful to know the actual number of bytes read and the reason the the read operation terminated. `IVisaReadResult` provides this information.

.NET DEFINITION

```
#if NET6_0_OR_GREATER
public interface IVisaReadResult
{
   Int64 ActualCount { get; }
   ReadStatus ReadStatus { get; }
}
#endif
```

9.2.2. Asynchronous I/O

VISA.NET Raw I/O asynchronous operations are implemented as a set of methods that allow a calling program to start an I/O operation and then do other tasks while waiting for I/O to complete.

The interface supports both the Event-Based Asynchronous Pattern (EAP) and Task-based Asynchronous Pattern (TAP). EAP requires multiple calls to complete an asynchronous operation. The TAP methods defined in this section require only one call to initiate and complete an asynchronous operation. Note that the TAP methods are omitted from the .NET Framework API.

Raw I/O includes methods that begin write and read operations, but return without waiting to see if the operations have completed. Several techniques may be used to check the status of that I/O operation, and to get the results when the I/O operation is complete.

ASYNCHRONOUS BEHAVIOR

Depending on the implementation, only one operation per resource may be allowed at a time, or several may be allowed at a time. If several asynchronous I/O operations for a resource are allowed at once, they are processed in the order in which they are initiated, so that reads and writes happen in a predictable order.

EAP I/O METHODS

EAP I/O starts with a call to a `BeginWrite` or `BeginRead` method. These methods return a reference to the `IVisaAsyncResult` interface. The interface includes information that uniquely identifies the operation, and can be used to communicate status and results.

EAP I/O can be aborted by calling `AbortAsyncOperation` Note that these methods take an `IVisaAsyncResult` argument that identifies the particular asynchronous I/O operation to abort.

EAP I/O operations are officialy completed by calling `EndWrite`, `EndRead`, or `EndReadString`. Note that these methods also take an `IVisaAsyncResult` argument that identifies the particular asynchronous I/O operation to end.

The appropriate End method must be called whenever a Begin method executed and returned a valid reference to `IVisaAsyncResult`. End methods perform required clean-up and disposal functions, and the implementation is free to leak if the End method is not called by the user. Note that calling `AbortAsyncOperation` does not relieve the user of the need to call an End method.

DETERMINING WHEN EAP ASYNCHRONOUS I/O IS COMPLETE

There are three ways that a calling program can determine when an EAP operation is complete - polling, blocking waits, and callbacks. Once a program has returned from a `BeginWrite` or `BeginRead` method call, it can poll the `IVisaAsyncResult.IsCompleted` property, waiting until it is set to `true`. Once a program has returned from a `BeginWrite` or `BeginRead` method call, it can wait on the `IVisaAsyncResult.AsyncResultHandle` event handle. The program will block on that event handle until the EAP I/O operation completes. Finally, if the `BeginWrite` or `BeginRead` method call includes a callback argument, the the callback method is invoked (exactly once) when the EAP I/O operation completes.

EAP ASYNCHRONOUS I/O RESULTS

EAP I/O uses the `IVisaAsyncResult` interface to identify particular asynchronous I/O operations, and to communicate status and results. `IVisaAsyncResult` derives from the Microsoft class `System.IAsyncResult`.

TAP ASYNCHRONOUS I/O METHODS

TAP methods are self-contained – multiple methods are not required to complete an asynchronous operation. The TAP methods are `ReadAsync` and `WriteAsync`.

TAP I/O can be aborted using the `cancellationToken` parameter (refer to Microsoft documentation for details).

TAP I/O operations are complete when the method returns.

TAP ASYNCHRONOUS I/O RESULTS

The TAP I/O read methods use the `IVisaReadResult` interface to return the read status and actual count.

9.2.2.1. IVisaAsyncResult

DESCRIPTION

When a `BeginWrite` or `BeginRead` method is called, it creates a new object that implements `IVisaAsyncResult` and returns a reference to the interface to the calling program. The calling program may use the reference to track certain information about the asynchronous operation that was initiated by the `BeginWrite` or `BeginRead` call.

Though `IVisaAsyncResult` roughly corresponds to `ViJobID` in VISA C, IVI.NET synchronous operations are significantly different from VISA C. For this reason, `IVisaAsyncResult` is described in detail in this section.

`IVisaAsyncResult` derives from `IAsyncResult`, which is described in MSDN documentation.

`IVisaAsyncResult` includes the following useful properties inherited from `System.IAsyncResult`:

- `AsyncState` is typed as an `object`. This contains application specific state information regarding the operation that was supplied by an argument to the `BeginRead` or `BeginWrite` method.
- `AsyncWaitHandle` is a .NET `WaitHandle` that can be used to wait for the completion of the asynchronous operation.
- `IsCompleted` indicates whether the asynchronous operation has completed.

`IVisaAsyncResult` defines the following additional properties:

- `IsAborted` indicates whether the asynchronous operation was aborted by a call to `AbortAsyncOperation`.
- `Buffer` is a reference to an array of bytes that holds the data being written or read.
- Before the operation has completed, `Count` is unspecified. After the operation has completed, it is the number of bytes actually read or written.
- `Index` is the value of the `index` argument to the `BeginRead` or `BeginWrite` method that initiated the I/O operation. If the method did not include an `index` argument, the value is zero.

DEFINITION

```
public interface IVisaAsyncResult : IAsyncResult
{
   Boolean IsAborted { get; }
   Byte[] Buffer { get; }
   Int64 Count { get; }
   Int64 Index { get; }
}
```

IMPLEMENTATION

RULE 9.2.2

For a particular asynchronous operation, if `BeginWrite` or `BeginRead` is called with the `state` parameter specified, the value of `IAsyncResult.AsyncState` in the returned `IVisaAsyncResult` reference **SHALL** be the value of the `state` parameter. If `BeginWrite` or `BeginRead` is called without the `state` parameter, the value of `IAsyncResult.AsyncState` **SHALL** be `Null`.

RULE 9.2.3

For a particular asynchronous read operation, `IVisaAsyncResult.Buffer` is unspecified until the operation has completed successfully. Once the operation has completed successfully, `Buffer` **SHALL** contain the bytes (starting at `Index`, if specified) that were read from the instrument.

RULE 9.2.4

For a particular asynchronous write operation, `IVisaAsyncResult.Buffer` **SHALL** contain the bytes that will actually be written to the instrument. For overloads of `BeginWrite` that take string buffer arguments, `IVisaAsyncResult.Buffer` contains the equivalent ASCII string after it has been converted from UNICODE.

RULE 9.2.5

For a particular asynchronous read or write operation, `IAsyncResult.AsyncWaitHandle` **SHALL** be signaled after the operation completes.

9.2.2.2. **AbortAsyncOperation**

DESCRIPTION

Requests the session to terminate normal execution of an asynchronous read or write operation.

Note that the associated asynchronous operation is considered to be complete after it has been aborted.

If the associated asynchronous operation was completed before it could be aborted by this method, it is not considered to have been aborted, even though this method was called. In this case, the method does not throw an exception, and the calling program must examine the `result.IsAborted` property to determine whether the operation completed successfully or not.

.

DEFINITION

```
void AbortAsyncOperation(IVisaAsyncResult result);
```

ARGUMENTS

Name	Description	Type
`result`	The reference to the pending asynchronous request to abort. The meaning of `IVisaAsyncResult` members after the call completes is listed below:	`IVisaAsyncResult`
`.AsyncState`	Unaffected by this method.	`Object`
`.AsyncWaitHandle`	Always signaled.	`WaitHandle`
`.CompletedSynchronously`	Unaffected by this method.	`Boolean`
`.IsCompleted`	Always *true* after this method completes.	`Boolean`
`.IsAborted`	Set to *true* if this method aborted the associated asynchronous operation, otherwise *false* (if the asynchronous operation was completed before it could be aborted by this method).	`Boolean`
`.Buffer`	Undefined.	`Byte[]`
`.Count`	Undefined	`Int64`
`.Index`	Unaffected by this method.	`Int64`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. Errors that occur during an asynchronous read request, such as an instrument communication failure during the IO request, occur on the thread pool thread and the corresponding exception will be thrown upon a call to `EndRead` or `EndWrite`.

9.2.2.3. **BeginRead**

DESCRIPTION

Begins an asynchronous read.

The calling program must call `EndRead` exactly once for every call to `BeginRead`. Failing to end an asynchronous operation before beginning another one can cause undesirable behavior such as a memory leak.

DEFINITION

```
IVisaAsyncResult BeginRead(Int32 count);
IVisaAsyncResult BeginRead(Int32 count, Object state);
IVisaAsyncResult BeginRead(Int32 count, VisaAsyncCallback callback,
                           Object state);

IVisaAsyncResult BeginRead(Byte[] buffer);
IVisaAsyncResult BeginRead(Byte[] buffer, Object state);
IVisaAsyncResult BeginRead(Byte[] buffer, Int64 index, Int64 count);
IVisaAsyncResult BeginRead(Byte[] buffer, Int64 index, Int64 count,
                           Object state);
IVisaAsyncResult BeginRead(Byte[] buffer, VisaAsyncCallback callback,
                           Object state);
IVisaAsyncResult BeginRead(Byte[] buffer, Int64 index, Int64 count,
                           VisaAsyncCallback callback, Object state);
```

ARGUMENTS

Name	Description	Type
`buffer`	The buffer to read data into. For overloads that include `buffer`, the calling program is expected to allocate the buffer. For overloads that do not include `buffer`, the implementation of this method allocates the buffer.	`Byte[]`
`count`	The maximum number of bytes to read. For overloads that do not include `count`, the default is the buffer size.	`Int64`
`index`	The byte offset in `buffer` at which to begin writing the data read. For overloads that do not include `index`, the default is 0.	`Int64`
`callback`	The method to be called when the asynchronous read operation is completed. Overloads that do not include `callback` leave it to the calling program to check for completion. .	`IVisaAsyncCallback`
`state`	A reference to an object that contains arbitrary information of interest to the calling program, and related to the asynchronous operation. This allows the asynchronous operation to provide the reference back to the calling program (as context) when the operation is complete. This is particularly useful when callbacks are used. Note that the object is not used by the asynchronous operation.	`Object`

RETURNS

Return Value	Description	Type
return value	An object that implements `IVisaAsyncResult`, which represents the status of an asynchronous operation. The object is constructed by this method. The meaning of `IVisaAsyncResult` members in this context is listed below:	`IVisaAsyncResult`
`.AsyncState`	A reference to the state object passed as a parameter to this method. This will be `null` if no state object was passed to this method.	`Object`
`.AsyncWaitHandle`	A handle that can be used to wait for the read to complete.	`WaitHandle`
`.CompletedSynchronously`	*true* if the read operation completed synchronously, otherwise *false*.	`Boolean`
`.IsCompleted`	*true* if the asynchronous operation is complete, otherwise *false*.	`Boolean`
`.IsAborted`	`IsAborted` is always *false* when returned by this method since this method begins the asynchronous read operation.	`Boolean`
`.Buffer`	Undefined.	`Byte[]`
`.Count`	Undefined.	`Int64`
`.Index`	The value of the `index` argument passed to this method. The value is zero if the method does not take an `index` argument.	`Int64`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. Errors that occur during an asynchronous read request, such as an instrument communication failure during the IO request, occur on the thread pool thread and become visible upon a call to `EndRead`.

IMPLEMENTATION

PERMISSION 9.2.1

If a call by a session class to the callback method fails, the failure **MAY** be ignored.

OBSERVATION 9.2.3

VISA.NET I/O implementations should not assume a client callback will return in any timeframe. Calling programs may execute blocking waits from callbacks. If a VISA.NET I/O resource component calls a callback, and the callback blocks in the same thread on which the callback was called, the callback will not return until the block completes.

OBSERVATION 9.2.4

The `callback` and `state` parameters are allowed to be `null`.

RULE 9.2.6

If `CompletedSynchronously` is `true` when this method returns, `IsCompleted` **SHALL** also be `true`.

OBSERVATION 9.2.5

If `CompletedSynchronously` is `false`, then depending on the timing of the I/O, it is possible for `IsCompleted` to be true when this method returns.

RULE 9.2.7

If a callback method is specified, the implementation **SHALL** call the callback exactly once for every successful call to `BeginRead`.

RULE 9.2.8

If a callback method is specified, the implementation **SHALL** allow the callback to call `EndRead` or `EndReadString`.

RULE 9.2.9

If a buffer is specified, the implementation of BeginRead **SHALL NOT** resize it.

9.2.2.4. **BeginWrite**

DESCRIPTION

Begins an asynchronous write.

The calling program must call `EndWrite` exactly once for every call to `BeginWrite`. Failing to end an asynchronous operation before beginning another one can cause undesirable behavior such as a memory leak.

DEFINITION

```
IVisaAsyncResult BeginWrite(String buffer);
IVisaAsyncResult BeginWrite(String buffer, Object state);
IVisaAsyncResult BeginWrite(String buffer, VisaAsyncCallback callback,
                            Object state);

IVisaAsyncResult BeginWrite(Byte[] buffer);
IVisaAsyncResult BeginWrite(Byte[] buffer, Object state);
IVisaAsyncResult BeginWrite(Byte[] buffer, Int64 index, Int64 count);
IVisaAsyncResult BeginWrite(Byte[] buffer, Int64 index, Int64 count,
                            Object state);
IVisaAsyncResult BeginWrite(Byte[] buffer, VisaAsyncCallback callback,
                            Object state);
IVisaAsyncResult BeginWrite(Byte[] buffer, Int64 index, Int64 count,
                            VisaAsyncCallback callback, Object state);
```

ARGUMENTS

Name	Description	Type
`buffer`	The buffer from which data is written.	`Byte[]` `String`
`count`	The maximum number of bytes to write. For overloads that do not include `count`, the default is the buffer size.	`Int64`
`index`	The byte offset in `buffer` at which to begin reading the data to be written. For overloads that do not include `index`, the default is 0.	`Int64`
`callback`	The method to be called when the asynchronous write operation is completed. Overloads that do not include `callback` leave it to the calling program to check for completion.	`VisaAsyncCallback`
`state`	A reference to an object that contains arbitrary information of interest to the calling program. This allows the asynchronous operation to provide the reference back to the calling program when the operation is complete. Note that the object is not used by the asynchronous operation.	`Object`

RETURNS

Return Value	Description	Type

return value		An object that implements `IVisaAsyncResult`, which represents the status of an asynchronous operation. The object is constructed by this method. The meaning of `IVisaAsyncResult` members in this context is listed below:	`IVisaAsyncResult`
	`.AsyncState`	A reference to the state object passed as a parameter to this method. This will be `null` if no state object was passed to this method.	`Object`
	`.AsyncWaitHandle`	A handle that can be used to wait for the write to complete.	`WaitHandle`
	`.CompletedSynchronously`	*true* if the write operation completed synchronously, otherwise *false*.	`Boolean`
	`.IsCompleted`	*true* if the asynchronous operation is complete, otherwise *false*.	`Boolean`
	`.IsAborted`	`IsAborted` is always *false* when returned by this method since this method begins the asynchronous write operation.	`Boolean`
	`.Buffer`	The buffer that was passed into the call to `BeginWrite` that initiated this asynchronous operation, or a buffer that includes only the bytes that will actually be written by this asynchronous operation.	`Byte[]`
	`.Count`	Undefined	`Int64`
	`.Index`	The value of the `index` argument passed to this method. The value is zero if the method does not take an `index` argument.	`Int64`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. Errors that occur during an asynchronous write request, such as an instrument communication failure during the IO request, occur on the thread pool thread and become visible upon a call to `EndWrite`.

IMPLEMENTATION

RULE 9.2.10

All `BeginWrite` methods that write a string **SHALL** convert the .NET string passed in to an ASCII string. If there is a UNICODE character that has an ambiguous or no conversion to ASCII, the method **SHALL** throw `an exception`.

PERMISSION 9.2.2

If a call by a session class to the callback method fails, the failure **MAY** be ignored.

OBSERVATION 9.2.6

VISA.NET I/O implementations should not assume a client callback will return in any timeframe. Calling programs may execute blocking waits from callbacks. If a VISA.NET I/O resource component calls a callback, and the callback blocks in the same thread on which the callback was called, the callback will not return until the block completes.

OBSERVATION 9.2.7

The `callback` and `state` parameters are allowed to be `null`.

RULE 9.2.11

If `CompletedSynchronously` is `true` when this method returns, `IsCompleted` **SHALL** also be `true`.

OBSERVATION 9.2.8

If `CompletedSynchronously` is `false`, then depending on the timing of the I/O, it is possible for `IsCompleted` to be true when this method returns.

RULE 9.2.12

If a callback method is specified, the implementation **SHALL** call the callback exactly once for every successful call to `BeginWrite`.

RULE 9.2.13

If a callback method is specified, the implementation **SHALL** allow the callback to call `EndWrite`.

9.2.2.5. **EndRead**

DESCRIPTION

Waits for the pending asynchronous read to complete. This method is always blocking.

DEFINITION

```
Int64 EndRead(IVisaAsyncResult result);
String EndReadString(IVisaAsyncResult result);
```

ARGUMENTS

The following table reflects the value of the result parameter upon completion of this method.

Name	Description	Type
`result`	The reference to the asynchronous request. The meaning of `IVisaAsyncResult` members after the call completes is listed below:	`IVisaAsyncResult`
`.AsyncState`	Unaffected by this method.	`Object`
`.AsyncWaitHandle`	Not specified after this method completes.	`WaitHandle`
`.CompletedSynchronously`	Unaffected by this method.	`Boolean`
`.IsCompleted`	`IsCompleted` is always *true* when returned by this method since this method waits until the asynchronous operation ends.	`Boolean`
`.IsAborted`	Unaffected by this method.	`Boolean`
`.Buffer`	A buffer is a valid buffer whose contents depends on which overload of `BeginRead` was called for this asynchronous operation.	`Byte[]`
`.Count`	The number of bytes read by this ansynchronous operation.	`Int64`
`.Index`	Unaffected by this method.	`Int64`

RETURN VALUE

Name	Description	Type
Return value	The number of bytes read by this ansynchronous operation into`IVisaAsyncResult.Buffer`.	`Int64`
	`IVisaAsyncResult.Buffer` converted to a Unicode string.	`String`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. In addition, any errors detected during the asynchronous operation will be thrown as exceptions from this method.

This method uses the `Ivi.Visa.IoTimeoutException` to report a timeout.

IMPLEMENTATION

RULE 9.2.14

The `EndReadString` method **SHALL** convert the entire contents of IVisaAsyncResult.Buffer to a .NET string. If there is an ambiguous conversion to ASCII, the method **SHALL** throw an exception.

OBSERVATION 9.2.9

The intent of EndReadString is to convert an entire response from the instrument to a string. If the overload of `BeginRead` used to initiate the operation used a user-allocated buffer that is larger than the number of bytes read, the results may be unexpected.

9.2.2.6. **EndWrite**

DESCRIPTION

Waits for the pending asynchronous write to complete. This method is always blocking.

DEFINITION

```
void EndWrite(IVisaAsyncResult result);
```

ARGUMENTS

The following table reflects the value of the result parameter upon completion of this method.

Name	Description	Type
`result`	The reference to the asynchronous request. The meaning of `IVisaAsyncResult` members after the call completes is listed below:	`IVisaAsyncResult`
`.AsyncState`	Unaffected by this method.	`Object`
`.AsyncWaitHandle`	Not specified after this method completes.	`WaitHandle`
`.CompletedSynchronously`	Unaffected by this method.	`Boolean`
`.IsCompleted`	`IsCompleted` is always *true* when returned by this method since this method waits until the asynchronous operation ends.	`Boolean`
`.IsAborted`	Unaffected by this method.	`Boolean`
`.Buffer`	The buffer that was passed into the call to `BeginWrite` that initiated this asynchronous operation, or a buffer that includes only the bytes that were actually to be written by this asynchronous operation.	`Byte[]`
`.Count`	The number of bytes actually written by this operation.	`Int64`
`.Index`	Unaffected by this method.	`Int64`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. In addition, any errors detected during the asynchronous operation will be thrown as exceptions from this method.

This method uses the `Ivi.Visa.IoTimeoutException` to report a timeout.

OBSERVATION 9.2.10

The `Buffer` value returned as part of the `IVisaAsyncResult` return value for this method is the same as the `Buffer` value returned as part of the `IVisaAsyncResult` return value for `BeginWrite()`.

9.2.2.7. **ReadAsync**

DESCRIPTION

Performs a TAP asynchronous read.

DEFINITION

```
#if NET6_0_OR_GREATER
Task<IVisaReadResult> ReadAsync(
                    Memory<Byte> buffer,
                    CancellationToken token = default);

Task<IVisaReadResult> ReadAsync(
                    Byte[] buffer,
                    Int64 index = 0,
                    Int64 count = -1,
                    CancellationToken token = default);
#endif
```

ARGUMENTS

Name	Description	Type
`buffer`	The buffer to read data into. The calling program is expected to allocate the buffer.	`Byte[]` `Memory<Byte>`
`index`	The byte offset in `buffer` at which to begin writing the data read. If the buffer is Memory<Byte>, the index is assumed to be 0.	`Int64`
`count`	The maximum number of bytes to read. If the buffer is Memory<Byte>, the count is assumed to be the size of the memory object. If count is -1, `count` defaults to the buffer size.	`Int64`
`token`	A token that can be used to cancel the asynchronous operation if needed.	`CancellationToken`

RETURNS

Return Value	Description	Type
`return value`	An interface reference that includes the actual number of bytes read and the reason for termination.	`IVisaReadResult`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. Errors that occur during an asynchronous read request, such as an instrument communication failure during the IO request, are returned from the `ReadAsync` method per TAP pattern conventions.

IMPLEMENTATION

OBSERVATION 9.2.11

VISA.NET I/O implementations should not assume an awaited call to `ReadAsync` will return in any timeframe. The implementation may use callbacks that are not visible to the client, and those callbacks may execute blocking waits. If the callback blocks in the same thread on which the callback was called, the callback will not return until the block completes.

RULE 9.2.15

The implementation of `ReadAsync` **SHALL NOT** resize the buffer.

9.2.2.8. **WriteAsync**

DESCRIPTION

Performs a TAP asynchronous write.

DEFINITION

```
#if NET6_0_OR_GREATER
Task WriteAsync(
                   ReadOnlyMemory<Byte> buffer,
                   CancellationToken token = default);

Task WriteAsync(Byte[] buffer,
                   Int64 index = 0,
                   Int32 count = -1,
                   CancellationToken token = default);
#endif
```

ARGUMENTS

Name	Description	Type
`buffer`	The buffer from which data is written.	`Byte[]` `String`
`index`	The byte offset in `buffer` at which to begin reading the data to be written. For overloads that do not include `index`, the default is 0.	`Int64`
`count`	The maximum number of bytes to write. For overloads that do not include `count`, the default is the buffer size. If count is -1, `count` defaults to the buffer size.	`Int64`
`token`	A token that can be used to cancel the asynchronous operation if needed.	`CancellationToken`

RETURNS

Return Value	Description	Type
return value	A Task object that allows the method to be used with the await keyword.	`Task`

EXCEPTIONS

Some exceptions (such as argument exceptions) are thrown immediately from this method. Errors that occur during an asynchronous write request, such as an instrument communication failure during the IO request, are returned from the `WriteAsync` method per TAP pattern conventions.

IMPLEMENTATION

OBSERVATION 9.2.12

VISA.NET I/O implementations should not assume an awaited call to `WriteAsync` will return in any timeframe. The implementation may use callbacks that are not visible to the client, and those callbacks may execute blocking waits. If the callback blocks in the same thread on which the callback was called, the callback will not return until the block completes.

9.3. Custom Formatting

When using the VISA.NET formatted I/O `Printf` and `Scanf` methods, the bulk of formatting and parsing is accomplished with the standard format specifiers. These specifiers work with common, simple types like strings, integers, and floating point numbers. Furthermore, these *standard conversions* are tightly specified, and the IVI Foundation provides a standard implementation for each one.

However, `Printf` and `Scanf` only have format specifiers for a few common .NET data types. Most .NET data types do not have format specifiers, and of course there are no format specifiers for user-defined types. When `Printf` and `Scanf` are called upon to format or parse a data type for which no format specifier exists, the end user must provide a *custom conversion* to do the job. A *type formatter* is a .NET class that allows `Printf` and `Scanf` to perform these custom conversions directly.

Example: Custom Formatting Challenge

Suppose an instrument accepts a SCPI command for setting the trigger source. Such a command might look something like the following:

```
TRIGger:SOURce [EXTernal|INTernal|SOFTware]
```

The three different values for the trigger source could be very naturally represented in programming languages by an enumeration, such as the following C# enumeration:

```
public enum TriggerSource
{
    External,
    Internal,
    Software
}
```

However, there is no format specifier to convert the enumeration values to the values used in the instrument's SCPI command. A custom conversion is needed to format and parse the instrument values.

In VISA C and VISA COM, custom conversions must be done in the calling program either before calling `Printf` (when formatting) or after calling `Scanf` (when parsing results), since both `Printf` and `Scanf` deal nicely with strings using standard format specifiers. However, VISA.NET provides the infrastructure for performing custom conversions inside of the `Printf` and `Scanf` methods themselves, which makes the calling program cleaner, isolates the custom conversions in a class designed just for that purpose, and allows `Printf` and `Scanf` to do a better job at the tasks for which they were designed (formatting and parsing, respectively).

9.3.1. Type Formatters

INTRODUCTION

A VISA.NET type formatter class (or *type formatter*) implements whatever logic is necessary to perform custom conversions between custom types and strings inside of `Printf` and `Scanf`.

Imagine a type formatter named `MyFormatter` that converts the TriggerSource enumeration values External, Internal, and Software (from the example in the previsous section) to the strings "EXT", "INT", and "SOFT". The code to use this formatter in a VISA.NET client would look something like this (where `session` is the VISA.NET session):

```
TriggerSource source = TriggerSource.Internal;
session.FormattedIO.TypeFormatter = (ITypeFormatter)MyFormatter;
session.FormattedIO.Printf("TRIG:SOUR %s\n", source);
```

When the VISA.NET implementation encounters the `%s` format specification in the above call, it recognizes that the associated parameter (`source`) is not a data type that it natively understands. It then

looks to see if an object that implements `ITypeFormatter` has been associated with the session. Since that was done in line two, VISA.NET invokes the `ITypeFormatter.IsSupported` method with the `source` argument to see if that type is supported by the type formatter. If it is, then the `ITypeFormatter.ToString` method is invoked to convert the `TriggerSource` value to the corresponding string. If it is not supported or if a type formatter has not been associated with the session, the VISA.NET implementation throws `ArgumentException`.

IMPLEMENTATION

OBSERVATION 9.3.1

The TypeFormatter is used with the %s and %c formatters.

OBSERVATION 9.3.2

For Printf, if the format specifier is "%s" (or "%,s") and the corresponding argument is type String, the string argument is used directly. If TypeFormatter is not null and TypeFormatter.IsSupported = true, Printf will use TypeFormatter to format the string; otherwise Printf will throw an exception.

OBSERVATION 9.3.3

For Scanf, if the format specifier is "%s" (or "%,s") the type formatter is consulted if it is not null. If TypeFormatter.IsSupported = true, Scanf will use the TypeFormatter to parse the string. If the TypeFormatter is null, and the argument supplied is a string, then the data scanned is returned directly in the user supplied argument. If the TypeFormatter is null or TypeFormatter.IsSupported = false, and the argument supplied is not a string, then a format exception is thrown.

OBSERVATION 9.3.4

Type formatters must implement the standard `ITypeFormatter` interface.

- A type formatter is associated with a VISA.NET formatted I/O session by assigning its `ITypeFormatter` interface to the `TypeFormatter` property in the `IMessageBasedFormattedIO` interface.
- Once this assignment is done, the `Printf` and `Scanf` methods will use the `ITypeFormatter` interface of the type formatter assigned to the `TypeFormatter` property as they format and parse instrument strings.

OBSERVATION 9.3.5

There is no single, standard type formatter.

- Different applications may use different formats for the same .NET type. For example, one application may format Boolean values as "true" and "false", another as "0" and "1", and another as "ON" and "OFF"
- Many types are application specific. For example, an enumeration whose members denote the channels of a particular instrument will be specific to applications that connect to that instrument.

OBSERVATION 9.3.6

In general, type formatters are application specific. VISA.NET users should be prepared to create one or more custom type formatters for their applications if they choose to use the type formatting features of VISA.NET.

OBSERVATION 9.3.7

Since only one type formatter can be associated with a formatted I/O session at a time (the `TypeFormatter` property is a scalar), a type formatter must be capable of performing all of the "custom" conversions that a single `Printf` or `Scanf` method call might encounter. An implementor may choose to implement all of the custom conversions needed for a particular application or component in a single type formatter, or implement them in several type formatters - that implementation decision is left to developers.

CORRESPONDING VISA FEATURES

VISA.NET type formatters have no corresponding feature in VISA C or VISA COM.

9.3.2. ITypeFormatter Interface

DESCRIPTION

The `ITypeFormatter` interface provides methods that perform custom conversions of supported .NET types to and from a string. It also provides a method for determining whether a particular type is supported.

DEFINITION

```
public interface ITypeFormatter
{
   Boolean IsSupported(Type type);
   String ToString(Object obj);
   Object Parse(Type type, String data);
}
```

CORRESPONDING VISA FEATURES

The `ITypeFormatter` interface methods have no corresponding functions in VISA C or VISA COM.

9.3.2.1. **IsSupported**

DESCRIPTION

Returns `true` if `type` is supported by the type formatter object. If `true`, the object must support converting a type value to a string (formatting) and converting a string to a type value (parsing).

DEFINITION

```
Boolean IsSupported(Type type);
```

ARGUMENTS

Name	Description	Type
`type`	The type which is tested to determine if it is supported by the formatter object.	`Type`

RETURNS

Return Value	Description	Type
return value	`true` if `type` is supported by the type formatter object, otherwise `false`.	`Boolean`

9.3.2.2. **ToString**

DESCRIPTION

Returns a string representation of `obj` that is suitable for use in a string that is formatted by `Printf`.

DEFINITION

```
String ToString(Object obj);
```

ARGUMENTS

Name	Description	Type
`obj`	The object to be formatted as a string.	`Object`

RETURNS

Return Value	Description	Type
return value	The string to which the object has been formatted.	`String`

EXCEPTIONS

This method throws the `Ivi.Visa. TypeFormatterException` when `obj` cannot be formatted as a string by the type formatter object.

9.3.2.3. **Parse**

DESCRIPTION

Returns an object of type `type`, the value of which is determined by parsing the `data` string.

DEFINITION

```
Object Parse(Type objectType, String data);
```

ARGUMENTS

Name	Description	Type
`type`	The data type of the returned object.	`Type`
`data`	The string data which is parsed to create the object returned by this method. The type formatter object must be able to parse `data` and convert it to an object of type `type`.	`String`

RETURNS

Return Value	Description	Type
return value	An object of type `type`, the value of which is determined by parsing the `data` string.	`Object`

EXCEPTIONS

This method throws the `Ivi.Visa. TypeFormatterException` when `data` cannot be parsed to an object of type `type` by the type formatter object.

9.4. IMessageBasedFormattedIO

DESCRIPTION

This section summarizes `IMessageBasedFormattedIO`. Note that `IMessageBasedFormattedIO` allows calling programs to use a variety of common data types. Given the necessary formatting direction, formatted I/O methods format and parse instrument string or buffer data appropriately.

`IMessageBasedFormattedIO` is all synchronous.

`IMessageBasedSession` provides a property that returns a reference to `IMessageBasedFormattedIO`. This property is the recommended way to access the `IMessageBasedFormattedIO` interface from a message-based session.

DEFINITION

The `IMessageBasedFormattedIO` interface declaration is shown below. The body of the interface is documented in the sections that document individual properties and methods.

```
public interface IMessageBasedFormattedIO
```

Refer to section 9.5, FormattedIO Implementations, for information about formatted I/O implementation options, and the standard IVI implementation of `IMessageBasedFormattedIO` in particular.

CORRESPONDING VISA FEATURES

The `IMessageBasedFormattedIO` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IMessageBasedFormattedIO`.

Property Name	VISA Attribute Name
`BinaryEncoding`	N/A
`ReadBufferSize`	VI_ATTR_RD_BUF_SIZE
`WriteBufferSize`	VI_ATTR_WR_BUF_SIZE
`TypeFormatter`	N/A

The `IMessageBasedFormattedIO` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IMessageBasedFormattedIO`.

Method Name	VISA Method Name
`DiscardBuffers`	`viFlush`
`FlushWrite`	`viFlush`
`Skip, SkipUntilEnd`	`viScanf with %* modifier`
`Printf`	`viPrintf`
`PrintfAndFlush`	
`PrintArray`	
`PrintArrayAndFlush`	
`Scanf`	`viScanf`
`ScanfArray`	
`Write`	`viPrintf` with specific format specifiers.
`WriteLine`	`viPrintf` with specific format specifiers.
`WriteList`	`viPrintf` with specific format specifiers.

`WriteLineList`	`viPrintf` with specific format specifiers.
`WriteBinary`	`viPrintf` with specific format specifiers.
`WriteBinaryAndFlush`	`viPrintf` with specific format specifiers.
`Read`*`Type`*	`viScanf` with specific format specifiers.
`ReadLine` `ReadLine`*`Type`*	`viScanf` with specific format specifiers.
`ReadListOf`*`Type`*	`viScanf` with specific format specifiers.
`ReadLineListOf`*`Type`*	`viScanf` with specific format specifiers.
`ReadBinaryBlockOf`*`Type`*	`viScanf` with specific format specifiers.
`ReadLineBinaryBlockOf`*`Type`*	`viScanf` with specific format specifiers.
`ReadWhileMatch, ReadUntilMatch, ReadUntilEnd`	`viScanf` with specific format specifiers.

IMPLEMENTATION

OBSERVATION 9.4.1

Most of the properties and methods in `IMessageBasedFormattedIO` have corresponding attributes and functions in VISA C, but the properties and methods in `IMessageBasedFormattedIO` differ, some slightly and some significantly, from the corresponding VISA C attributes and functions. For this reason, all of the methods in this interface are described in detail below.

9.4.2. BinaryEncoding

DESCRIPTION

The binary encoding used by Write and Read methods when formatting or parsing an array of numeric data.

DEFINITION

```
BinaryEncoding BinaryEncoding { get; set; }
```

IMPLEMENTATION

The formatting and parsing rules associated with each of the enumerated values for BinaryEncoding correspond to Printf and Scanf format specifiers as shown in the following table.

Value	Format Specifier
`DefiniteLengthBlockData`	%b
`IndefiniteLengthBlockData`	%B
`RawLittleEndian`	%!oly
`RawBigEndian`	%!oby

9.4.3. ReadBufferSize

DESCRIPTION

The size of the internal formatted I/O read buffer.

DEFINITION

```
Int32 ReadBufferSize { get; set; }
```

9.4.4. WriteBufferSize

DESCRIPTION

The size of the internal formatted I/O write buffer.

DEFINITION

```
Int32 WriteBufferSize { get; set; }
```

9.4.5. TypeFormatter

DESCRIPTION

A reference to the `ITypeFormatter` interface implemented by the type formatter that will be used by `Printf` and `Scanf` to format and parse the types that it supports.

DEFINITION

```
ITypeFormatter TypeFormatter { get; set; }
```

9.4.6. DiscardBuffers

DESCRIPTION

Discards all of the data in both the formatted I/O read and write buffers, but does not send anything to the instrument.

DEFINITION

```
void DiscardBuffers();
```

9.4.7. FlushWrite

DESCRIPTION

Flushes all the data from the write buffer and sends it to the instrument.

If an exception occurs during this method, the buffer will be cleared. The calling program does not need to explicitly call `FlushWrite` or `DiscardBuffers` before attempting another Write or Printf operation.

DEFINITION

```
void FlushWrite(Boolean sendEnd);
```

IMPLEMENTATION

RULE 9.4.2

The `FlushWrite`method **SHALL** send the buffer to the instrument with END if `sendEnd` is `true`, otherwise it **SHALL** send the buffer without END.

RULE 9.4.3

If the `SendEndEnabled` property is different from the value of the sendEnd parameter, the FlushWrite method **SHALL** change it on the I/O session, commit the write buffer, and then restore it.

RULE 9.4.4

FlushWrite method **SHALL NOT** change the characters in the buffer. For example, it will not add a termchar to the contents of the buffer.

9.4.8. Printf Format Strings

The `Printf` method formats data provided by input arguments. The data is created by replacing each format specifier in the `format` parameter with the corresponding parameter argument formatted according to the format specifier. The data is formatted as ASCII strings, IEEE-488.2 arbitrary blocks, and raw binary blocks.

9.4.8.1. Printf Format Argument

The `format` argument consists of ordinary characters, and format specifiers. Format specifiers describe the format in which associated input arguments are to be written. When the string is written to the device, the formatted data is substituted for the format specifier. Any UNICODE character may be used in a format argument, as long as the formatted string can be converted to ASCII.

When a newline (0x000A) is encountered in the format string, the following actions are taken (in order):

- A newline is appended to the write buffer.
- All data is flushed from the write buffer and sent to the instrument with an END indicator.

OBSERVATION 9.4.2

The mechanism used to represent a newline (0x000A) in a format string is language dependent.

9.4.8.2. Printf Format Specifiers

`Printf` format strings may include one or more format specifiers, each of which provides information about how to format one of the variable arguments to `Printf`. Format specifiers for `Printf` are very similar to those for the VISA `viPrintf` function, though there are some differences. For example, 'a', 'A', 'C', 'n', 'p', and 'S' are not supported in .NET. Refer to *VPP-4.3: The VISA Library*, Section 6.2.3, *viPrintf(vi, writeFmt, arg1, arg2,...)*, for a description of format specifiers in VISA.

A basic format specifier always starts with a '%' (percent character) and ends with a format type, which indicates the data type of the value to be formatted. For example, "%d" is a format specifier that will print a signed integer as a sequence of decimal digits. If the integer is negative, it will be preceeded by a minus sign.

A variety of optional modifiers may be added to the basic format specifier to provide a rich set of formatting options. The general syntax of a VISA.NET format specifier is,

"**%**[flags][width][**.**precision][,array_size][size_modifier]type"

For example, "%+@3d" is a format specifier that will print a signed integer in scientific notation, with a plus sign if the integer is non-negative and a minus sign if it is negative. Note that some modifiers are not valid with some format types, and some modifiers have different meanings, depending on the type.

For each format specifier, there is one *value argument* that corresponds to the format specifier type, and there may be additional *dynamic arguments* (indicated by a '*****' in the specifier) that provide information about the format specifier's modifiers. In the argument list, the dynamic arguments for a format specifier always come before the value argument. Printf formats a value (from a value argument) using a format specifier that may need additional information from dynamic arguments to be complete. For example,

```
Printf("[%-*.*s]", 10, 20, "John");
```

will print the string "`[John      ]`". The first dynamic argument, 10, is the width - the minimum number of characters to print. The second dynamic argument, 20, is the precision – the maximum number of characters to print. If the string to be printed were longer than 20 characters, only twenty would be printed. The '-' indicates that if the string is shorter than the number of characters to be printed, the string should be left justified and, by default, padded with spaces.

The following sections describe the format specifier types and modifiers in more detail:

- Format types

- Flags
- Width, Precision, and Array Size
- Size Modifiers

FORMAT TYPES

Every format specifier has a format type that indicates the data type of the corresponding value argument, and by extension determines that the format will be appropriate to the data. In VISA.NET, format types may correspond to several .NET data types. In general, if the corresponding value argument can be correctly formatted using a format specifier, it will be.

Note that since the .NET version of `Printf` can determine the data types of all of the input arguments, format types are not needed for that purpose, as they are in VISA. Format types are only needed to help describe the desired format. However, if the type of a corresponding value argument is not compatible with the format type, `Printf` will throw an exception.

The following table lists the format types recognized by `Printf`, along with a basic description of the type and the valid .NET data types for the corresponding value argument.

Printf Format Specifier Types		
Types	**Format Corresponding Input Argument As**	**Valid Value Argument Types**
Characters and Strings		
`c`	An ASCII character. If the input argument is a String or StringBuilder variable, only the first character is formatted.	`Char`, `String` (Must contain exactly one character)
`s`	An ASCII string.	`String`
Integer Numbers (formatted as strings)		
`d, i, u`	An integer formatted as an ASCII string. (Unsigned types are cast to Int64 and then formatted. UInt64 numbers greater that Int64.MaxValue will be formatted as negative numbers.)	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[], Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[]`
`o`	An unsigned integer formatted as an octal ASCII string. Signed integer values are treated as unsigned for formatting.	
`x`	An unsigned integer formatted as a hexadecimal ASCII string. Signed integer values are treated as unsigned for formatting. Digits ‘a’-‘f’ are lowercase. If there is a “0x” prefix, the x is also lowercase.	
`X`	An unsigned integer formatted as a hexadecimal ASCII string. Signed integer values are treated as unsigned for formatting. Digits ‘A’-‘F’ are uppercase. If there is a “0X” prefix, the X is also uppercase.	
Real Numbers (formatted as strings)		
`e`	A real number formatted as an ASCII string using scientific notation. The ‘e’ that introduces the exponent is lowercase.	`Single, Single[], Double, Double[]`
`E`	A real number formatted as an ASCII string using scientific notation. The ‘E’ that introduces the exponent is uppercase.	`Single, Single[], Double, Double[]`
`f`	A real number formatted as an ASCII string using arithmetic notation.	`Single, Single[], Double, Double[]`

g	A real number formatted as an ASCII string using arithmetic or scientific notation, depending on the scale of the number. If scientific notation is used, the 'e' that introduces the exponent is lowercase.	`Single, Single[], Double, Double[]`
G	A real number formatted as an ASCII string using arithmetic or scientific notation, depending on the scale of the number. If scientific notation is used, the 'E' that introduces the exponent is lowercase.	`Single, Single[], Double, Double[]`
IEEE 488.2 Blocks		
b	An array of numbers formatted as an IEEE-488.2 definite length block. The sign is not specified. A size modifier must be specified to properly format the block. The size modifier must match the type of the corresponding array argument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[],Single[], Double[]` (`Int64[] and UInt64[]` are not supported at this time.)
B	An array of numbers formatted as an IEEE-488.2 indefinite length block. The sign is not specified. A size modifier must be specified to properly format the block. The size modifier must match the type of the corresponding array argument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[],Single[], Double[]` (`Int64[] and UInt64[]` are not supported at this time.)
Raw Binary		
y	An array of signed or unsigned integers formatted as a binary array. The array may be specified as big endian or little endian. A size modifier must be specified to properly format the array. The size modifier must match the type of the corresponding array argument. Byte order may be specified for raw binary arrays by using "!ol" (little Endian) or "!ob" (big Endian) immediately after the '%' character that introduces the specifier. The default is big Endian.	`Byte[], SByte[], Int16[],UInt16[], Int32[], UInt32[], Int64[], UInt64[]`
Not Valid for Printf: *'t', 'T'*		
Not Valid in VISA.NET: *'a', 'A', 'C', 'n', 'p', 'S'*		

FLAGS

Flags are optional characters or strings that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. Flags immediately follow the '%' character that begins the format specifier. More than one flag can appear in a format specification. Format specifiers that include an invalid flag, an invalid combination of flags, or flags that are out of order should not be used. The results are undefined and Printf may throw an exception or return arbitrary results.

VISA.NET recognizes several ANSI defined flags (space, '+', '0', '-', and '#') and several VISA specific flags that support IEEE 488.2 formatting ('@1', '@2', '@3', '@H', '@Q', and '@B'). The following general rules are observed when determining what combinations of flags are valid in a format specifier.

1. A particular flag may only be used once in a format specifier.
2. You can't use a '#' flag and one of the IEEE '@' flags in the same format specifier, because every combination inherently conflicts.
3. The formatting specified by an IEEE '@' flag overrides the default formatting of the format specifier type.
4. Only one '@' flag may be included in a format specifier.
5. If a format specifier is not listed next to an "@" flag in the table below, the results are undefined, may throw an exception, and should not be used.
6. The ANSI flags (space, '+', '0', '-', and '#') are order independent, but precede the '@' flags.

Additional, more specific rules are also noted in the table below.

Flag	Valid For Types	Description
'-'	c, s d, i, o, u, x, X e, E, f, g, G	Left align the formatted string within the given field width. Note that '-' is only valid if the width is specified. If the '-' flag is not included, the formatted string is right aligned within the given field width.
' ' (space)	d, i, o, u, x, X e, E, f, g, G	Prefix the output value with a space if the output value is signed and positive; if the format specifier includes both a space flag and a '+' flag, the space is ignored. The space flag is ignored if the specifier also includes one of the IEEE '@H', '@Q', or '@B' flags.
'0'	d, i, o, u, x, X e, E, f, g, G	Pad the output value with the '0' character. Note that '0'is ignored if the width is not specified, or the specifier also includes the '-' flag or one of the IEEE '@H', '@Q', or '@B' flags.
'+'	d, i, o, u, x, X e, E, f, g, G	Prefix the output value(s) with a sign (+ or –). If the '+' flag is not included, the sign is only printed if the integer is negative. The '+' flag is ignored if the specifier also includes one of the '@H', '@Q', or '@B' flags.
'Q', 'q'	s	Enclose strings in double ('Q') or single ('q') quotes. For string arrays, individual elements are enclosed in double or single quotes.
'@1'	d, i, u e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 NR1 format. This is the default format for format types d, i, and u. For real types, the real is truncated before formatting.
'@2'	d, i, u e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 NR2 format. This is the default format for format type f.
'@3'	d, i, u e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 NR3 format. This is the default format for format type E.

‘@H‘	d, i, u, x, X e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 <HEXADECIMAL_NUMERIC_RESPONSE_DATA> format. For real types, the real is truncated before formatting.
‘@Q‘	d, i, o, u e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 <OCTAL_NUMERIC_RESPONSE_DATA> format. For real types, the real is truncated before formatting.
‘@B‘	d, i, o, u, x, X e, E, f, g, G	The output value(s) are formatted in IEEE_488.2 <BINARY_NUMERIC_RESPONSE_DATA> format. For real types, the real is truncated before formatting.

The VISA “#” flag is not currently recognized in VISA.NET, but will be added at a future date.

WIDTH, PRECISION, AND ARRAY SIZE MODIFIERS

The width, precision, and array size modifiers are all integers that describe either the size of the space in which a value is to be formatted, or the number of array elements to be formatted. The values may be explicitly given as part of the format specifier, or they may be implicitly indicated by a '*' character, and filled in from the `Printf` argument list. These modifiers follow the format specifier flags.

Modifier	**Valid For Types**	**Description**
width	c, s d, i, o, u, x, X e, E, f, g, G	Optional. The minimum width of the formatted value. The string is padded with spaces on the left if '-' is not specified, and on the right if '-' is specified. If an array is specified, width applies to each element individually. If width is '*', then the value of width is read from an input argument. The argument preceeds the precision and array_size input arguments, if they are specified, and the value input argument. The width input argument may be a signed or unsigned positive integer. For other values (zero, fractions, negative), the results are undefined, may throw an exception, and should not be used.
.precision	c, s d, i, o, u, x, X e, E, f, g, G	Optional. For integer types (d, i, o, u, x, X): The maximum width of the printed value. If the full formatted value is longer than <precision> characters, the first precision characters are printed and the rest are discarded. For the real types e and f: The actual number of digits after the decimal point For real type g: The actual number of significant digits.For type s: The maximum number of characters printed. For type c: The precision is ignored. If an array is specified, precision applies to each element individually. If precision is '*', then the value of precision is read from an input argument. The argument preceeds the array_size input argument, if there is one, and the value input argument. The precision input argument may be a signed or unsigned positive integer. For other values (zero, fractions, negative), the results are undefined, may throw an exception, and should not be used.

,array_size	d, i, o, u, x, X e, E, f, g, G	Optional. The ‘,’ character indicates an array of numbers, optionally followed by the number of elements to be formatted. It is only necessary to include array_size if the number of elements to be printed is less than the number of elements in the input array argument. Arrays indicated by the ‘,’ modifier are formatted as comma separated lists, with each element formatted according to the format specifier. The array size is determined as follows: • If *array_size* is ‘*’, then the value of array size is read from an input argument. The argument preceeds the value input argument. The *array_size* input argument may be a signed or unsigned positive integer. • If *array_size* is a positive integer, that is the array size. • If *array_size* is not included in the format specifier, *array_size* is derived automatically from the number of elements in the corresponding argument if it is an array, otherwise the behavior is undefined. • If *array_size* is less than or equal to 0, greater than the size of the associated array, or fractional, , the results are undefined, may throw an exception, and should not be used.

SIZE MODIFIERS

Size modifiers indicate the size of the data to be formatted.

When formatting numbers as ASCII strings, VISA.NET, unlike VISA or the standard version of printf(), does not need size modifiers to determine the size of the value argument. As a result, size modifiers are ignored when formatting numbers as ASCII strings.

When formatting character and strings, size modifiers are invalid.

Size modifiers are required when formatting IEEE-488.2 arbitrary blocks and raw binary arrays. In these cases, the type of the array argument to Printf that corresponds to the format specifier must match the size modifier. For example, if the format specifier is "%ly", the corresponding argument must be an array of 32-bit integers.

Modifier	Valid For Types	Description
none	b, B, y	8-bit integers (the default for b and B)
h	b, B, y	16-bit integers
l	b, B, y	32-bit integers
ll	y	64-bit integers
z	b, B	32-bit reals in IEEE 754 format.
Z	b, B	64-bit reals in IEEE 754 format.

For size modifiers not listed in the above table, the results are undefined, may throw an exception, and should not be used.

9.4.8.3. **Printf Format Specifier Usage Summary**

The printf method uses a regular expression to verify each format specifier type. The regular expression is,

```
@"
(?<literalChars> [^%]+|(%%)) |

(?<number> % \s*
        (?<flags> (\+ ((\-0)|(0?\-?))?) | (\- ((\+0)|(0?\+?))?) | (0 ((\+\-)|(\-?\+?))?) )? \s*
        (?<IeeeType> @[123HQB])? \s*
        (?<width> \d+|\*)? \s*
        (\. (?<precision> \d+|\*))? \s*
        (?<sizeModifier> [hlL]|(ll))? \s*
        (?<typeCode> [dDiIoOuUxXfeEgG])) |

(?<numberList> %  \s*
        (?<flags> (\+ ((\-0)|(0?\-?))?) | (\- ((\+0)|(0?\+?))?) | (0 ((\+\-)|(\-?\+?))?) )? \s*
        (?<IeeeType> @[123HQB])? \s*
        (?<width> \d+|\*)? \s*
        (\. (?<precision> \d+|\*))? \s*
        (?<delimiter> ,) \s*
        (?<length> \d+|\*)? \s*
        (?<sizeModifier> [hlL]|(ll))? \s*
        (?<typeCode> [dDiIoOuUxXfeEgG])) |

(?<binaryBlock> % \s*
        (?<length> \d+|\*)? \s*
        (?<sizeModifier> [hlLzZ])? \s*
        (?<typeCode> [bB])) |

(?<rawBinary> % \s*
        (?<length> \d+|\*)? \s*
        (?<byteOrder> (!ol) | (!ob) )? \s*
        (?<sizeModifier> [hl]|(ll))? \s*
        (?<typeCode> y)) |

(?<char> % \s*
        (?<flags> (\+\-)|(\-\+?))?  \s*
        (?<width> \d+|\*)? \s*
        (\. (?<precision> \d+|\*))? \s*
        (?<typeCode> c)) |

(?<string> % \s*
        (?<flags> (\+\-)|(\-\+?))?  \s*
        (?<width> \d+|\*)? \s*
        (\. (?<precision> \d+|\*))? \s*
        (?<quotes>q|Q)? \s*
        (?<typeCode> s)) |

(?<stringList> % \s*
        (?<flags> (\+\-)|(\-\+?))?  \s*
        (?<width> \d+|\*)? \s*
        (\. (?<precision> \d+|\*))? \s*
        (?<delimiter> ,) \s*
        (?<length> \d+|\*)? \s*
        (?<quotes>q|Q)? \s*
        (?<typeCode> s))”
```

9.4.9. Printf

DESCRIPTION

Writes formatted data to the formatted write buffer. The data is created by replacing each format specifier in the format parameter with the corresponding parameter argument formatted according the format specifier. The data is formatted as ASCII strings, IEEE-488.2 arbitrary blocks, and raw binary blocks.

DEFINITION

```
void Printf(String data);
void Printf(String format, params object[] args);
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`data`	The string literal to be printed.	`String`
`args[]`	A variable number of arguments, each of which is either • data to be formatted into the format string using the corresponding format specifiers in the format string, or • width, precision, or array size values to be substituted for occurrences of '*' in format specifiers	`object`

IMPLEMENTATION

RULE 9.4.5

Printf **SHALL** throw appropriate exceptions for the following conditions:

- The `format` argument is `null` or an empty string.
- One or more of the format specifiers in `format` is not supported.
- One or more of the format specifiers is not valid.
- One or more of the format specifiers does not match the data type of the corresponding input argument.
- The number of format specifiers exceeds the number of input arguments

RULE 9.4.6

All characters placed in the formatted I/O write buffer **SHALL** be formatted as ASCII strings, IEEE-488.2 blocks, or raw binary blocks. All characters copied directly from the format parameter **SHALL** be formatted as ASCII strings.

RULE 9.4.7

IF `Printf` fails to write a character to the write buffer because it cannot convert the character to an ASCII character, it **SHALL** throw an exception that describes the problem and identifies the character.

RULE 9.4.8

For IEEE-488.2 block format specifier types 'b', 'B', and 'y', **IF** the type of the input argument array does not match the size modifier of the block format specifier, the method **SHALL** throw an ArgumentException exception.

9.4.10. PrintfAndFlush

DESCRIPTION

The behavior for `PrintfAndFlush` is the same as `Printf` followed by a `FlushWrite` with `sendEnd` equal to `true`.

If an exception occurs during this method, the buffer will be cleared. The calling program does not need to explicitly call `FlushWrite` or `DiscardBuffers` before attempting another Write or Printf operation.

DEFINITION

```
void PrintfAndFlush(String data);
void PrintfAndFlush(String format, params object[] args);
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`data`	The string literal to be printed.	`String`
`args[]`	A variable number of arguments, each of which is either • data to be formatted into the format string using the corresponding format specifiers in the format string, or • width, precision, or array size values to be substituted for occurrences of '*' in format specifiers.	`object`

9.4.11. PrintfArray

DESCRIPTION

Writes formatted numeric array data to the formatted write buffer without requiring the calling program to make a copy of the data. The behavior for `PrintfArray` is the same as `Printf` for a single array.

DEFINITION

```
unsafe void PrintfArray(String format, Byte* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, SByte* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, Int16* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, UInt16* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, Int32* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, UInt32* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, Int64* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, UInt64* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, Single* pArray, params Int64[] inputs);
unsafe void PrintfArray(String format, Double* pArray, params Int64[] inputs);
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`pArray`	A pointer to an array of numbers.	`Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double`
`inputs`	A variable number of integer arguments consisting of width, precision, or array size values to be substituted for occurrences of '*' in format specifiers	`Int64[]`

9.4.12. PrintfArrayAndFlush

DESCRIPTION

The behavior for `PrintfArrayAndFlush` is the same as `PrintfArray` followed by a `WriteFlush` with `sendEnd` equal to `true`.

If an exception occurs during this method, the buffer will be cleared. The calling program does not need to explicitly call `FlushWrite` or `DiscardBuffers` before attempting another Write or Printf operation.

DEFINITION

```
unsafe void PrintfArrayAndFlush(String format, Byte* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, SByte* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, Int16* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, UInt16* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, Int32* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, UInt32* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, Int64* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, UInt64* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, Single* pArray,
                                params Int64[] inputs);
unsafe void PrintfArrayAndFlush(String format, Double* pArray,
                                params Int64[] inputs
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`pArray`	A pointer to an array of numbers.	`Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double`
`inputs`	A variable number of integer arguments consisting of width, precision, or array size values to be substituted for occurrences of '*' in format specifiers	`Int64[]`

9.4.13. Scanf Format Strings

The `Scanf` method reads and parses data into input arguments. The parsing is done by examining the input data for element(s) that correspond to each format specifier in the `format` parameter, and then storing the results in the corresponding output parameter arguments. The input data is formatted as ASCII strings, IEEE-488.2 arbitrary blocks, and raw binary blocks.

9.4.13.1. Scanf Format Argument

The `format` argument consists of ordinary characters and format specifiers. Format specifiers describe the format from which associated input arguments are to be parsed. When the string is read from the device, the format specifier is used to parse the data that corresponds to the format specifier. Any UNICODE character that has an ASCII equivalent may be used in a format argument.

When a newline (0x000A) is encountered in the format string, either the newline is considered to be a whitespace character or there must be a corresponding newline in the read buffer, depending on the context.

OBSERVATION 9.4.3

The mechanism used to represent a newline in a format string is language dependent.

9.4.13.2. Scanf Format Specifiers

`Scanf` format strings may include one or more format specifiers, each of which provides information about how to parse the input and extract a value for one of the variable arguments to `Scanf`. Format specifiers for `Scanf` are very similar to those for the VISA `viScanf` function, though there are some differences. For example, 'a', 'A', 'C', 'n', 'p', and 'S' are not supported in .NET. Refer to *VPP-4.3: The VISA Library*, Section 6.2.8, *viScanf(vi, readFmt, arg1, arg2,...)*, for a description of format specifiers in VISA.

A basic format specifier always starts with a '%' (percent character) and ends with a format type, which indicates the data type of the value to be parseed. For example, "%d" is a format specifier that will read a signed integer as a sequence of decimal digits. If the integer is negative, it will be preceeded by a minus sign.

A variety of optional modifiers may be added to the basic format specifier to provide a rich set of parsing options. The general syntax of a VISA.NET format specifier is,

"**%**[flags] [width] [,array_size] [size_modifier] type"

For example, "%+@3d" is a format specifier that will read a signed integer in scientific notation, with a plus sign if the integer is non-negative and a minus sign if it is negative. Note that some modifiers are not valid with some format types, and some modifiers have different meanings, depending on the type.

For each format specifier, there is one *value argument* that corresponds to the format specifier type, and there may be additional *modifier arguments* (indicated by a '#' in the specifier) that provide information about the format specifier's modifiers. In the argument list, the modifier arguments for a format specifier are elements in the `inputs` argument. `Scanf` parses a value (from a value argument) using a format specifier that may need additional information from modifier arguments to be complete. For example,

```
Scanf("[%-#s]", {10}, "John");
```

will read the string "John". The first valiable argument, 10, is the width – the maximum number of characters to be read for this format specifier. If the input string in the formatted read buffer were longer than 10 characters, only ten would be read.

The following sections describe the format specifier types and modifiers in more detail:

- Format types
- Flags
- Width and Array Size Modifiers
- Size Modifiers

FORMAT TYPES

Every format specifier has a format type that indicates the data type of the corresponding output argument, and by extension determines that the format of the input data that is appropriate to the corresponding value argument. In VISA.NET, format types may correspond to several .NET data types. In general, if the corresponding value argument can be correctly determined using a format specifier, it will be.

Note that since the VISA.NET version of `Scanf` can determine the data types of all of the input arguments, format types are not needed for that purpose, as they are in VISA. Format types are only needed to help describe the desired format. However, if the type of a corresponding value argument is not compatible with the format type, `Scanf` will throw an exception.

The following table lists the format types recognized by `Scanf`, along with a basic description of the type and the valid .NET data types for the corresponding value argument.

Scanf Format Specifier Types		
Types	**Parse Corresponding Input As**	**Valid Value Argument Types**
Characters and Strings		
`c`	An ASCII character.	`Char, String`
`s`	An ASCII string.	`String`
`[<m>]`	An ASCII string consisting of characters that match characters in the string <m>.	`String`
`[^<m>]`	An ASCII string consisting of characters that match characters not in the string <m>.	`String`
`t/T`	An ASCII string.	`String`
Integer Numbers (formatted as strings)		
`d`	A signed integer formatted as a decimal ASCII string. When used to scan a floating point number, the number is rounded to the nearest integer according to IEEE 488.2 rules.	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[],Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[],Object`
`i`	An integer formatted as an ASCII string. It may be formatted as a decimal, octal, or hexadecimal string. When used to scan a floating point number, the number is rounded to the nearest integer according to IEEE 488.2 rules.	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[],Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[],Object`
`o`	An unsigned integer formatted as an octal ASCII string. When used to scan a floating point number, the number is rounded to the nearest integer according to IEEE 488.2 rules.	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[],Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[], Object`
`u`	An unsigned integer formatted as an ASCII string. When used to scan a floating point number, the number is rounded to the nearest integer according to IEEE 488.2 rules.	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[],Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[], Object`

x/X	An unsigned integer formatted as a hexadecimal ASCII string. When used to scan a floating point number, the number is rounded to the nearest integer according to IEEE 488.2 rules.	`SByte, SByte[], Int16, Int16[], Int32, Int32[], Int64, Int64[],Byte, Byte[], UInt16, UInt16[], UInt32, UInt32[], UInt64, UInt64[], Object`
Real Numbers (formatted as strings)		
e/E	A real number formatted as an ASCII string using scientific notation.	`Single, Single[], Double, Double[]`
f	A real number formatted as an ASCII string using arithmetic notation.	`Single, Single[], Double, Double[]`
g/G	A real number formatted as an ASCII string using arithmetic or scientific notation.	`Single, Single[], Double, Double[]`
IEEE 488.2 Blocks		
b	An array of integers formatted as an IEEE-488.2 block. The sign is not specified. A size modifier must be specified to properly format the block. The size modifier must match the type of the corresponding array argument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[],Single[], Double[]` (`Int64[] and UInt64[]` are not supported at this time.)
Raw Binary		
y	An array of signed or unsigned integers formatted as a binary array. The array may be specified as big endian or little endian. A size modifier must be specified to properly format the array. The size modifier must match the type of the corresponding array argument. Byte order may be specified for raw binary arrays by using "!ol" (little Endian) or "!ob" (big Endian) immediately after the '%' character that introduces the specifier. The default is big Endian.	`Byte[], SByte[], Int16[],UInt16[], Int32[], UInt32[], Int64[], UInt64[]`
Not Valid for Scanf: *'a', 'A', 'B'*		
Not Valid in VISA.NET: *'C', 'n', 'p', 'S'*		

For type 's', initial whitespace characters (including newline) are discarded. Characters starting with the first non-whitespace character are read into the output argument until the first whitespace character or an END indicator is found. The whitespace character is not included in the output argument, and does not remain in the buffer. If an END indicator is found on a non-whitespace character, that character is removed from the buffer and returned.

For type '[<m>]', characters are read into the output argument until the first character not included in <m> or an END indicator is found. The character not included in <m> is not included in the output argument, and is not removed from the buffer. If an END indicator is found on a matching character, that character is removed from the buffer and returned. The <m> token may include character ranges such as "0-9".

For type '[^<m>]', characters are read into the output argument until the first character included in <m> or an END indicator is found. The character included in <m> is not included in the output argument, and is not removed from the buffer. If an END indicator is found on a non-matching character, that character is removed from the buffer and returned.

For type ’t’, characters are read into the output argument until the END indicator is found. The character on which the END indicator was received is included in the output argument, and is removed from the buffer.

For type ‘T’, characters are read into the output argument until the first newline character or an END indicator is found. The newline character is included in the output argument, and is removed from the buffer. If an END indicator is found on a non-newline character, that character is removed from the buffer and returned.

For integer format specifier types, if the corresponding argument is typed as an unsigned integer and the number being scanned is negative, an exception is thrown.

For numeric types, initial whitespace characters (including newlines) are read and discarded. Then characters are read until a character that cannot be interpreted as part of the number is encountered. That character remains in the buffer.

OBSERVATION 9.4.4

The implementation uses RegEx.IsMatch() when using ‘[<m>]’, which does recognize ranges. Other RegEx matching characters are also recognized by Scanf.

FLAGS

Flags are optional characters or strings that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. Flags immediately follow the '%' character that begins the format specifier. More than one flag can appear in a format specification. Format specifiers that include an invalid flag, an invalid combination of flags, or flags that are out of order should not be used. The results are undefined and Scanf may throw an exception or return arbitrary results.

VISA.NET recognizes one ANSI defined flag ('*') and several VISA specific flags that support parsing IEEE 488.2 formats ('@1', '@2', '@3', '@H', '@Q', and '@B'). The following general rules are observed when determining what combinations of flags are valid in a format specifier.

1. The formatting specified by an IEEE '@' flag overrides the default formatting of the format specifier type.
2. If a format specifier is not listed next to an "@" flag in the table below, the results are undefined, may throw an exception, and should not be used.
3. Only one '@' flag may be included in a format specifier.

Flag	Valid For Types	Description
'*'	c, s, t, T [<m>], [^<m>] d, i, o, u, x, X e, E, f, g, G b, y	The field in the input that corresponds to this format specifier is read but not stored in an output argument.
'Q', 'q'	s	Strings are enclosed in quotes. For string arrays, individual elements are enclosed in quotes. The quotes are stripped when returning the scanned value.
'@1'	d, i, u	The input value(s) are formatted as a number. There is not necessarily an expectation that the number is formatted in IEEE_488.2 NR1 format.
'@2'	d, i, u e, E, f, g, G	The input value(s) are formatted as a number. There is not necessarily an expectation that the number is formatted in IEEE_488.2 NR2 format.
'@3'	d, i, u e, E, f, g, G	The input value(s) are formatted as a number. There is not necessarily an expectation that the number is formatted in IEEE_488.2 NR3 format.
'@H'	d, i, u, x, X	Optional. The input value(s) are formatted in IEEE_488.2 <HEXADECIMAL_NUMERIC_RESPONSE_DATA> format.
'@Q'	d, i, o, u	Optional. The input value(s) are formatted in IEEE_488.2 <OCTAL_NUMERIC_RESPONSE_DATA> format.
'@B'	d, i, o, u, x, X	Optional. The input value(s) are formatted in IEEE_488.2 <BINARY_NUMERIC_RESPONSE_DATA> format.

WIDTH AND ARRAY SIZE MODIFIERS

The width and array size modifiers are all integers that describe either the size of the space in which a value is to be formatted, or the number of array elements to be formatted. The values may be explicitly given as part of the format specifier, or they may be implicitly indicated by a '#' character, and filled in from a value in the `Scanf` inputs parameter. These modifiers follow the format specifier flags.

Modifier	Valid For Types	Description
width	c, s, t, T [<m>], [^<m>] d, i, o, u, x, X e, E, f, g, G	Optional. The maximum number of characters to be parsed for this specifier. Fewer than *width* characters may be read if a whitespace character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached. If *width* is '#', then the value of *width* is read from a value in the `Scanf inputs` parameter. The *width* input argument may be a signed or unsigned positive integer. For other values (zero, fractions, negative), the results are undefined, may throw an exception, and should not be used.
,array_size	d, i, o, u, x, X e, E, f, g, G b, y	Optional. The ',' character indicates an array of numbers, optionally followed by the number of elements to be read. It is only necessary to include *array_size* if the number of elements to be read is less than the number of elements in the corresponding output array argument. Arrays indicated by the ',' modifier are formatted as comma separated lists. The array size is determined as follows: • If *array_size* is '#', then the value of *array_size* is read from a value in the `Scanf inputs` parameter. • If *array_size* is a positive integer, that is the array size. • If *array_size* is less than or equal to 0, or fractional , the results are undefined, may throw an exception, and should not be used.

SIZE MODIFIERS

Size modifiers indicate the size of the data to be read.

When reading numbers as ASCII strings, VISA.NET, unlike VISA or the standard version of scanf(), does not need size modifiers to determine the size of the value argument. As a result, size modifiers are ignored when parsing numbers as ASCII strings.

When reading character and strings, size modifiers are invalid.

Size modifiers are required when formatting IEEE-488.2 arbitrary blocks and raw binary arrays. In these cases, the type of the array argument to Scanf that corresponds to the format specifier must match the size modifier. For example, if the format specifier is "%ly", the corresponding argument must be an array of 32-bit integers.

Modifier	Valid For Types	Description
none	b, y	8-bit integers
h	b, y	16-bit integers
l	b, y	32-bit integers
ll	y	64-bit integers
z	b	32-bit reals
Z	b	64-bit reals

Size modifiers not listed in the above table are invalid if they are inherently inconsistent with the format type, and otherwise they are ignored.

9.4.13.3. Scanf Format Specifier Usage Summary

Regular expressions are used to parse each format specifier type. The regular expression is:

```
@"
(?<literalChars> [^%]+|(%%)) |

(?<number> % \s*
        (?<suppress> \*)?  \s*
        (?<IeeeType> @[123HQB])? \s*
        (?<width> \d+|\#)? \s*
        (?<sizeModifier> [hlL]|(ll))? \s*
        (?<typeCode> [dDiIoOuUxXfeEgG])) |

(?<numberList> %  \s*
        (?<suppress> \*)?  \s*
        (?<IeeeType> @[123HQB])? \s*
        (?<width> \d+|\#)? \s*
        (?<delimiter> ,) \s*
        (?<length> \d+|\#)? \s*
        (?<sizeModifier> [hlL]|(ll))? \s*
        (?<typeCode> [dDiIoOuUxXfeEgG])) |

(?<binaryBlock> % \s*
        (?<suppress> \*)?  \s*
        (?<length> \d+|\#)? \s*
        (?<sizeModifier> [hlLzZ])? \s*
        (?<typeCode> b)) |

(?<rawBinary> % \s*
        (?<suppress> \*)?  \s*
        (?<length> \d+|\#)? \s*
        (?<byteOrder> (!ol) | (!ob) )? \s*
        (?<sizeModifier> [hl]|(ll))? \s*
        (?<typeCode> y)) |

(?<char> % \s*
        (?<suppress> \*)?  \s*
        (?<width> \d+|\#)? \s*
        (?<typeCode> [cTt])) |

(?<stringNoWhitespace> % \s*
        (?<suppress> \*)?  \s*
        (?<width> \d+|\#)? \s*
        (?<quotes>q|Q)? \s*
        (?<typeCode> s)) |

(?<stringWithWhitespace> % \s*
        (?<suppress> \*)?  \s*
        (?<width> \d+|\#)? \s*
        (?<charSet> \[ \^? [^\]]* \])) |

(?<stringList> % \s*
        (?<suppress> \*)?  \s*
        (?<width> \d+|\#)? \s*
        (?<delimiter> ,) \s*
        (?<length> \d+|\#)? \s*
        (?<quotes>q|Q)? \s*
        (?<typeCode> s))”
```

9.4.14. Scanf

DESCRIPTION

Reads a formatted string from the formatted read buffer, and parses the string according to the specified format. The parsing process extracts typed values from the formatted string into `out` arguments, based on corresponding format specifiers in the format string.

DEFINITION

```
void Scanf<T>(String format,
              out T output);
void Scanf<T1, T2>(String format,
              out T1 output1, out T2 output2);
void Scanf<T1, T2, T3>(String format,
              out T1 output1, out T2 output2, out T3 output3);
void Scanf<T1, T2, T3, T4>(String format,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4);
void Scanf<T1, T2, T3, T4, T5>(String format,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5);
void Scanf<T1, T2, T3, T4, T5, T6>(String format,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5, out T6 output6);
void Scanf<T1, T2, T3, T4, T5, T6, T7>(String format,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5, out T6 output6, out T7 output7);

void Scanf<T>(String format, Int32[] inputs,
              out T output);
void Scanf<T1, T2>(String format, Int32[] inputs,
              out T1 output1, out T2 output2);
void Scanf<T1, T2, T3>(String format, Int32[] inputs,
              out T1 output1, out T2 output2, out T3 output3);
void Scanf<T1, T2, T3, T4>(String format, Int32[] inputs,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4);
void Scanf<T1, T2, T3, T4, T5>(String format, Int32[] inputs,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5);
void Scanf<T1, T2, T3, T4, T5, T6>(String format, Int32[] inputs,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5, out T6 output6);
void Scanf<T1, T2, T3, T4, T5, T6, T7>(String format, Int32[] inputs,
              out T1 output1, out T2 output2, out T3 output3, out T4 output4,
              out T5 output5, out T6 output6, out T7 output7);
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`inputs`	The values that are substituted for the '#' characters in format specifiers. Values are substituted in the order in which the '#' characters appear in the specifiers.	`Int32[]`

`output,`
`output1,`
`output2,`
`output3,`
`output4,`
`output5,`
`output6,`
`output7`
	A variable number of argumentsthat all represent values to be parsed from the format string using the corresponding format specifiers in the format string.	`Types listed in the format type table in Section 9.4.13.2, or supported by the registered type converter.`

IMPLEMENTATION

RULE 9.4.9

Scanf **SHALL** throw appropriate exceptions for the following format specifier errors:

- The `format` argument is `null` or an empty string.
- One or more of the format specifiers in format is not supported.
- One or more of the format specifiers in format is not valid.
- One or more of the format specifiers does not match the data type of the corresponding output argument.
- The number of format specifiers exceeds the number of output arguments

RULE 9.4.10

The `Scanf` operation accepts input until an END indicator is read or characters corresponding to the `format` argument (including all format specifiers) are read. Thus, detecting an END indicator before the `format` argument 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 `format` argument are satisfied, the data will be kept in the buffer and will be used by the next `Scanf` operation.

OBSERVATION 9.4.5

The raw I/O `Read` method is used for the actual low-level read from the device. Therefore, `Read` should not be used in the same session with formatted I/O operations, including `Scanf`. Also, if multiple sessions using formatted I/O resources are connected to the same device, the client is responsible for synchronizing the actual low-level reads.

OBSERVATION 9.4.6

Notice that when an END indicator is received, not all arguments in the `format` argument may be consumed. However, the operation still returns successfully, and the remaining unscanned output arguments are assigned the value default(T).

RULE 9.4.11 **(VISA - 6.2.11)**

The formatted I/O read operations **SHALL** honor the state of the `TerminationCharacterEnabled` property.

OBSERVATION 9.4.7 **(VISA - 6.2.9)**

Although formatted I/O operations generally read until an END indicator is received, RULE 9.4.11RULE 9.4.11 allows the user to also specify a termination character that, if read as part of string data, will cause the formatted I/O operations to stop reading from the device.

RULE 9.4.12

`Scanf` **SHALL** disable the termination character (if it is enabled) while reading data from a definate binary block, but must turn it back on before reading data lying outside the block.

RULE 9.4.13 **(VISA-COM 7.1.21)**

If a timeout occurs during a formatted read method, but enough data was retrieved to complete the request, the method **SHALL NOT** throw an exception.

OBSERVATION 9.4.8 **(VISA-COM OBS 7.1.1)**

A timeout can occur but the operation can still be successful if the END signal is suppressed and the termination character is disabled, in which case the only way to complete reading data of indefinite size is to encounter a timeout.

RULE 9.4.14 **(VISA - 6.2.15)**

IF the low level read operation used by `Scanf` times out,, but enough data was not retrieved to complete the request, **THEN** the formatted I/O read buffer **SHALL** be cleared before `Scanf` throws an exception.

OBSERVATION 9.4.9 **(VISA - 6.2.11)**

When the low level read operation used by `Scanf` times out, the next call to `Scanf` will read from an empty buffer and force a read from the device.

RULE 9.4.15**(VISA - 6.2.16)**

IF there is no remaining data to be parsed in the internal buffer, **AND** a new call to `Scanf` is made, **THEN** `Scanf` **SHALL** attempt to read more data from the instrument.

OBSERVATION 9.4.10 **(VISA - 6.2.11)**

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 `Scanf` with two numeric arguments versus two calls to `Scanf` each with one numeric argument. In the first case, OBSERVATION 9.4.6 points out that the single call will return VI_SUCCESS even though argument #2 is ignored. In the second case, RULE 9.4.15 points out that call #2 will not be ignored but will in fact read more data (or time out trying to do so).

OBSERVATION 9.4.11

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 remains in the internal buffer. If a user calls `Scanf` with a numeric argument such as `%d`, then the newline is treated as whitespace and is ignored. Thus, VISA will read more data. The format types to which Rule 9.4.8 applies are the string (s) and numeric (d, i, u, o, x, X, e, E, f, g, G) types (including lists). However, if a user calls `Scanf` 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. The rule 9.4.8 does not apply to the remaining format types (c, t, T, [], b, y).

9.4.15. ScanfArray

DESCRIPTION

Reads a formatted numeric array data from the formatted read buffer without requiring the calling program to make a copy of the data. The behavior for `ScanfArray` is the same as `Scanf` for a single array.

DEFINITION

```
unsafe Int64 ScanfArray(String format, Byte* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, SByte* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, Int16* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, UInt16* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, Int32* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, UInt32* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, Int64* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, UInt64* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, Single* pArray, params Int64[] inputs);
unsafe Int64 ScanfArray(String format, Double* pArray, params Int64[] inputs);
```

ARGUMENTS

Name	Description	Type
`format`	The format string, including all format specifiers.	`String`
`pArray`	A pointer to an array of numbers.	`Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double`
`inputs`	A variable number of arguments consisting of width, precision, or array size values to be substituted for occurrences of '*' in format specifiers	`Int64[]`

9.4.16. Introduction to Formatted Write Methods

Formatted write methods include `Write`, `WriteList`, `WriteLine`, `WriteLineList`, `WriteBinary`, and `WriteBinaryAndFlush` in the `IMessageBasedFormattedIO` interface. The section that describes each method also includes the equivalent `Printf` format specifier. To determine the equivalent behavior, refer to sections 9.4.8.3, Printf Format Specifier Usage Summary and 9.4.9, Printf for details.

9.4.17. Write

DESCRIPTION

Converts the specified data to an ASCII string and appends the resulting string to the write buffer.

DEFINITION

```
void Write(Char data);
void Write(String data);
void Write(Int64 data);
void Write(UInt64 data);
void Write(Double data);
```

ARGUMENTS

Name	Description	Type
`data`	A single character, string, integer, or real number to be converted to an ASCII string and sent to the instrument.	`Char, String, Int64, UInt64, Double`

PRINTF EQUIVALENTS

The `Write` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

Write Method	Equivalent Printf Call
`void Write(Char data);`	`Printf("%c"), data)`
`void Write(String data);`	`Printf("%s"), data)`
`void Write(Int64 data);`	`Printf("%@1d"), data)`
`void Write(UInt64 data);`	`Printf("%@1u"), data)`
`void Write(Double data);`	`Printf("%@2f"), data)`

9.4.18. WriteLine

DESCRIPTION

Performs the following operations in order:

- Converts the data specified to an ASCII string. The data is followed by a new line.
- Appends the resulting string to the write buffer.
- Writes the buffer to the instrument.
- Sends an END to the instrument if `SendEndEnabled` is `true`.
- Flushes the buffer.

DEFINITION

```
void WriteLine();
void WriteLine(Char data);
void WriteLine(String data);
void WriteLine(Int64 data);
void WriteLine(UInt64 data);
void WriteLine(Double data);
```

ARGUMENTS

Name	Description	Type
`data`	A single character, string, integer, or real number to be converted to an ASCII string and sent to the instrument.	`Char, String, Int64, UInt64, Double`

PRINTF EQUIVALENTS

The `WriteLine` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

WriteLine Method	Equivalent Printf Call
`void WriteLine(Char data);`	`Printf("%c\n"), data)`
`void WriteLine(String data);`	`Printf("%s\n"), data)`
`void WriteLine(Int64 data);`	`Printf("%@1d\n"), data)`
`void WriteLine(UInt64 data);`	`Printf("%@1u\n"), data)`
`void WriteLine(Double data);`	`Printf("%@2f\n"), data)`

9.4.19. WriteList

DESCRIPTION

Performs the following operations in order:

- Converts the data array specified to an ASCII string. Commas are placed between each element in the string.
- Appends the resulting string to the write buffer.

DEFINITION

```
void WriteList(Byte[] data);
void WriteList(Byte[] data, Int64 index, Int64 count);

void WriteList(SByte[] data);
void WriteList(SByte[] data, Int64 index, Int64 count);

void WriteList(Int16[] data);
void WriteList(Int16[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteList(UInt16[] data);
[CLSCompliant(false)]
void WriteList(UInt16[] data, Int64 index, Int64 count);

void WriteList(Int32[] data);
void WriteList(Int32[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteList(UInt32[] data);
[CLSCompliant(false)]
void WriteList(UInt32[] data, Int64 index, Int64 count);

void WriteList(Int64[] data);
void WriteList(Int64[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteList(UInt64[] data);
[CLSCompliant(false)]
void WriteList(UInt64[] data, Int64 index, Int64 count);

void WriteList(Single[] data);
void WriteList(Single[] data, Int64 index, Int64 count);

void WriteList(Double[] data);
void WriteList(Double[] data, Int64 index, Int64 count);
```

ARGUMENTS

Name	Description	Type

`data`	An array of numbers to be converted to an ASCII separated list sent to the instrument. The separator character is a comma.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element from `data` to be sent to the instrument.	`Int64`
`count`	The number of elements from `data` to be sent to the instrument, starting from `index`. `count` must be positive, and `index` + `count` <= `data.length`.	`Int64`

PRINTF EQUIVALENTS

The `WriteList` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

WriteList Method	**Equivalent Printf Call**

`void WriteList(Byte[] data);`
`void WriteList(UInt16[] data);`
`void WriteList(UInt32[] data);`
`void WriteList(UInt64[] data);`
	`Printf("%@1,u", data)`

`void WriteList(SByte[] data);`
`void WriteList(Int16[] data);`
`void WriteList(Int32[] data);`
`void WriteList(Int64[] data);`
	`Printf("%@1,d", data)`

`void WriteList(Single[] data);` `void WriteList(Double[] data);`	`Printf("%@2,f", data)`

9.4.20. WriteLineList

DESCRIPTION

Performs the following operations in order:

- Converts the data array specified to an ASCII string. Commas are placed between each element in the string. The data is followed by a new line.
- Appends the resulting string to the write buffer.
- Writes the buffer to the instrument.
- Sends an END to the instrument.
- Flushes the buffer.

DEFINITION

```
void WriteLineList(Byte[] data);
void WriteLineList(Byte[] data, Int64 index, Int64 count);

void WriteLineList(SByte[] data);
void WriteLineList(SByte[] data, Int64 index, Int64 count);

void WriteLineList(Int16[] data);
void WriteLineList(Int16[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteLineList(UInt16[] data);
[CLSCompliant(false)]
void WriteLineList(UInt16[] data, Int64 index, Int64 count);

void WriteLineList(Int32[] data);
void WriteLineList(Int32[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteLineList(UInt32[] data);
[CLSCompliant(false)]
void WriteLineList(UInt32[] data, Int64 index, Int64 count);

void WriteLineList(Int64[] data);
void WriteLineList(Int64[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteLineList(UInt64[] data);
[CLSCompliant(false)]
void WriteLineList(UInt64[] data, Int64 index, Int64 count);

void WriteLineList(Single[] data);
void WriteLineList(Single[] data, Int64 index, Int64 count);

void WriteLineList(Double[] data);
void WriteLineList(Double[] data, Int64 index, Int64 count);
```

ARGUMENTS

Name	Description	Type

`data`	An array of numbers to be converted to an ASCII separated list sent to the instrument. The separator character is a comma.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element from `data` to be sent to the instrument.	`Int64`
`count`	The number of elements from `data` to be sent to the instrument, starting from `index`. `count` must be positive, and `index` + `count` <= `data.length`.	`Int64`

PRINTF EQUIVALENTS

The `WriteLineList` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

WriteLineList Method	**Equivalent Printf Call**

`void WriteLineList(Byte[] data);`
`void WriteLineList(UInt16[] data);`
`void WriteLineList(UInt32[] data);`
`void WriteLineList(UInt64[] data);`
	`Printf("%@1,u\n", data)`

`void WriteLineList(SByte[] data);`
`void WriteLineList(Int16[] data);`
`void WriteLineList(Int32[] data);`
`void WriteLineList(Int64[] data);`
	`Printf("%@1,d\n", data)`

`void WriteLineList(Single[] data);` `void WriteLineList(Double[] data);`	`Printf("%@2,f\n", data)`

9.4.21. WriteBinary

DESCRIPTION

Performs the following operations in order:

- Converts the data array specified to a binary array. The `BinaryEncoding` property specifies whether to write a definite length IEEE-488.2 block, an indefinite length IEEE-488.2 block, a raw binary block in big endian format, or a raw binary block in little endian format.
- Appends the resulting data to the write buffer.

DEFINITION

```
void WriteBinary(Byte[] data);
void WriteBinary(Byte[] data, Int64 index, Int64 count);

void WriteBinary(SByte[] data);
void WriteBinary(SByte[] data, Int64 index, Int64 count);

void WriteBinary(Int16[] data);
void WriteBinary(Int16[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinary(UInt16[] data);
[CLSCompliant(false)]
void WriteBinary(UInt16[] data, Int64 index, Int64 count);

void WriteBinary(Int32[] data);
void WriteBinary(Int32[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinary(UInt32[] data);
[CLSCompliant(false)]
void WriteBinary(UInt32[] data, Int64 index, Int64 count);

void WriteBinary(Int64[] data);
void WriteBinary(Int64[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinary(UInt64[] data);
[CLSCompliant(false)]
void WriteBinary(UInt64[] data, Int64 index, Int64 count);

void WriteBinary(Single[] data);
void WriteBinary(Single[] data, Int64 index, Int64 count);

void WriteBinary(Double[] data);
void WriteBinary(Double[] data, Int64 index, Int64 count);
```

ARGUMENTS

Name	Description	Type

`data`	An array of numbers to be converted to a binary form (determined by `BinaryEncoding`) and placed in the output buffer.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element from `data` to be sent to the instrument.	`Int64`
`count`	The number of elements from `data` to be sent to the instrument, starting from `index`. `count` must be positive, and `index` + `count` <= `data.length`.	`Int64`

PRINTF EQUIVALENTS

The `WriteBinary` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

WriteLineList Method	**Equivalent Printf Call**
If BinaryEncoding = DefiniteLengthBlockData	
`void WriteBinary(Byte[] data);` `void WriteBinary(SByte[] data);`	`Printf("%b", data)`
`void WriteBinary(UInt16[] data);` `void WriteBinary(Int16[] data);`	`Printf("%hb", data)`
`void WriteBinary(UInt32[] data);` `void WriteBinary(Int32[] data);`	`Printf("%lb", data)`
`void WriteBinary(UInt64[] data);` `void WriteBinary(Int64[] data);`	`WriteBinary(UInt64[])` and `WriteBinary(Int64[])` for definite length blocks and the corresponding Printf format specifier (`%llb`) are not supported at this time.
`void WriteBinary(Single[] data);`	`Printf("%zb", data)`
`void WriteBinary(Double[] data);`	`Printf("%Zb", data)`
If BinaryEncoding = IndefiniteLengthBlockData	
`void WriteBinary(Byte[] data);` `void WriteBinary(SByte[] data);`	`Printf("%B", data)`
`void WriteBinary(UInt16[] data);` `void WriteBinary(Int16[] data);`	`Printf("%hB", data)`
`void WriteBinary(UInt32[] data);` `void WriteBinary(Int32[] data);`	`Printf("%lB", data)`
`void WriteBinary(UInt64[] data);` `void WriteBinary(Int64[] data);`	`WriteBinary(UInt64[])` and `WriteBinary(Int64[])` for indefinite length blocks and the corresponding Printf format specifier (`%llB`) are not supported at this time.
`void WriteBinary(Single[] data);`	`Printf("%zB", data)`
`void WriteBinary(Double[] data);`	`Printf("%ZB", data)`
If BinaryEncoding = RawBigEndian	
`void WriteBinary(Byte[] data);` `void WriteBinary(SByte[] data);`	`Printf("%!oby", data)`
`void WriteBinary(UInt16[] data);` `void WriteBinary(Int16[] data);`	`Printf("%!obhy", data)`

`void WriteBinary(UInt32[] data);` `void WriteBinary(Int32[] data);`	`Printf("%!obly", data)`
`void WriteBinary(UInt64[] data);` `void WriteBinary(Int64[] data);`	`Printf("%!oblly", data)`
`void WriteBinary(Single[] data);`	`WriteBinary(Single[])` for raw big endian arrays is supported. The corresponding Printf format specifier (%!obzy) is not supported at this time.
`void WriteBinary(Double[] data);`	`WriteBinary(Double[])` for raw big endian arrays is supported. The corresponding Printf format specifier (%!obZy) is not supported at this time.
If BinaryEncoding = RawLittleEndian	
`void WriteBinary(Byte[] data);` `void WriteBinary(SByte[] data);`	`Printf("%!oly", data)`
`void WriteBinary(UInt16[] data);` `void WriteBinary(Int16[] data);`	`Printf("%!olhy", data)`
`void WriteBinary(UInt32[] data);` `void WriteBinary(Int32[] data);`	`Printf("%!olly", data)`
`void WriteBinary(UInt64[] data);` `void WriteBinary(Int64[] data);`	`Printf("%!ollly", data)`
`void WriteBinary(Single[] data);`	`WriteBinary(Single[])` for raw little endian arrays is supported. The corresponding Printf format specifier (%!olzy) is not supported at this time.
`void WriteBinary(Double[] data);`	`WriteBinary(Double[])` for raw little endian arrays is supported. The corresponding Printf format specifier (%!olZy) is not supported at this time.

9.4.22. WriteBinary AndFlush

DESCRIPTION

Performs the following operations in order:

- Converts the data array specified to a binary aray. The `BinaryEncoding` property specifies whether to write a definite length IEEE-488.2 block, an indefinite length IEEE-488.2 block, a raw binary block in big endian format, or a raw binary block in little endian format.
- Appends the resulting data to the write buffer.
- Writes the buffer to the instrument.
- Sends an END to the instrument.
- Flushes the buffer.

DEFINITION

```
void WriteBinaryAndFlush(Byte[] data);
void WriteBinaryAndFlush(Byte[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(SByte[] data);
void WriteBinaryAndFlush(SByte[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(Int16[] data);
void WriteBinaryAndFlush(Int16[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt16[] data);
[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt16[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(Int32[] data);
void WriteBinaryAndFlush(Int32[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt32[] data);
[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt32[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(Int64[] data);
void WriteBinaryAndFlush(Int64[] data, Int64 index, Int64 count);

[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt64[] data);
[CLSCompliant(false)]
void WriteBinaryAndFlush(UInt64[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(Single[] data);
void WriteBinaryAndFlush(Single[] data, Int64 index, Int64 count);

void WriteBinaryAndFlush(Double[] data);
void WriteBinaryAndFlush(Double[] data, Int64 index, Int64 count);
```

ARGUMENTS

Name	Description	Type
`data`	An array of numbers to be converted to a binary form (determined by `BinaryEncoding`) and sent to the instrument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element from `data` to be sent to the instrument.	`Int64`
`count`	The number of elements from `data` to be sent to the instrument, starting from `index`. `count` must be positive, and `index` + `count` <= `data.length`.	`Int64`

PRINTF EQUIVALENTS

The `WriteBinaryAndFlush` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Printf`, as shown in the following table.

WriteLineList Method	Equivalent Printf Call
If BinaryEncoding = DefiniteLengthBlockData	
`void WriteBinaryAndFlush(Byte[] data);` `void WriteBinaryAndFlush(SByte[] data);`	`PrintfAndFlush("%b", data)`
`void WriteBinaryAndFlush(UInt16[] data);` `void WriteBinaryAndFlush(Int16[] data);`	`PrintfAndFlush("%hb", data)`
`void WriteBinaryAndFlush(UInt32[] data);` `void WriteBinaryAndFlush(Int32[] data);`	`PrintfAndFlush("%lb", data)`
`void WriteBinaryAndFlush(UInt64[] data);` `void WriteBinaryAndFlush(Int64[] data);`	`WriteBinaryAndFlush(UInt64[])` and `WriteBinaryAndFlush(Int64[])` for definite length blocks and the corresponding Printf format specifier (`%llb`) are not supported at this time.
`void WriteBinaryAndFlush(Single[] data);`	`PrintfAndFlush("%zb", data)`
`void WriteBinaryAndFlush(Double[] data);`	`PrintfAndFlush("%Zb", data)`
If BinaryEncoding = IndefiniteLengthBlockData	
`void WriteBinaryAndFlush(Byte[] data);` `void WriteBinaryAndFlush(SByte[] data);`	`PrintfAndFlush("%B", data)`
`void WriteBinaryAndFlush(UInt16[] data);` `void WriteBinaryAndFlush(Int16[] data);`	`PrintfAndFlush("%hB", data)`
`void WriteBinaryAndFlush(UInt32[] data);` `void WriteBinaryAndFlush(Int32[] data);`	`PrintfAndFlush("%lB", data)`
`void WriteBinaryAndFlush(UInt64[] data);` `void WriteBinaryAndFlush(Int64[] data);`	`WriteBinaryAndFlush(UInt64[])` and `WriteBinaryAndFlush(Int64[])` for indefinite length blocks and the corresponding Printf format specifier (`%llB`) are not supported at this time.
`void WriteBinaryAndFlush(Single[] data);`	`PrintfAndFlush("%zB", data)`
`void WriteBinaryAndFlush(Double[] data);`	`PrintfAndFlush("%ZB", data)`
If BinaryEncoding = RawBigEndian	
`void WriteBinaryAndFlush(Byte[] data);` `void WriteBinaryAndFlush(SByte[] data);`	`PrintfAndFlush("%!oby", data)`

`void WriteBinaryAndFlush(UInt16[] data);` `void WriteBinaryAndFlush(Int16[] data);`	`PrintfAndFlush("%!obhy", data)`
`void WriteBinaryAndFlush(UInt32[] data);` `void WriteBinaryAndFlush(Int32[] data);`	`PrintfAndFlush("%!obly", data)`
`void WriteBinaryAndFlush(UInt64[] data);` `void WriteBinaryAndFlush(Int64[] data);`	`PrintfAndFlush("%!oblly", data)`
`void WriteBinaryAndFlush(Single[] data);`	`WriteBinaryAndFlush(Single[])` for raw big endian arrays is supported. The corresponding Printf format specifier (%!obzy) is not supported at this time.
`void WriteBinaryAndFlush(Double[] data);`	`WriteBinaryAndFlush(Double[])` for raw big endian arrays is supported. The corresponding Printf format specifier (%!obZy) is not supported at this time.
If BinaryEncoding = RawLittleEndian	
`void WriteBinaryAndFlush(Byte[] data);` `void WriteBinaryAndFlush(SByte[] data);`	`PrintfAndFlush("%!oly", data)`
`void WriteBinaryAndFlush(UInt16[] data);` `void WriteBinaryAndFlush(Int16[] data);`	`PrintfAndFlush("%!olhy", data)`
`void WriteBinaryAndFlush(UInt32[] data);` `void WriteBinaryAndFlush(Int32[] data);`	`PrintfAndFlush("%!olly", data)`
`void WriteBinaryAndFlush(UInt64[] data);` `void WriteBinaryAndFlush(Int64[] data);`	`PrintfAndFlush("%!ollly", data)`
`void WriteBinaryAndFlush(Single[] data);`	`WriteBinaryAndFlush(Single[])` for raw little endian arrays is supported. The corresponding Printf format specifier (%!olzy) is not supported at this time.
`void WriteBinaryAndFlush(Double[] data);`	`WriteBinaryAndFlush(Double[])` for raw little endian arrays is supported. The corresponding Printf format specifier (%!olZy) is not supported at this time.

9.4.23. Introduction to Formatted Read Methods

Formatted read methods include `Read<Type>`, `ReadList<Type>`, `ReadLine<Type>`, `ReadLineList<Type>`, `ReadBinaryBlock<Type>`, and `ReadLineBinaryBlock<Type>`, `ReadWhileMatch`, `ReadUntilMatch`, and `ReadUntilEnd` in the `IMessageBasedFormattedIO` interface. The section that describes each method also includes the equivalent Scanf format specifier. To determine the equivalent behavior, refer to sections 9.4.13.3, Scanf Format Specifier Usage Summary, and 9.4.14, Scanf, for details.

9.4.24. ReadString

DESCRIPTION

Reads a string from the formatted read buffer.

DEFINITION

```
String ReadString();
String ReadString(Int32 count);
Int32 ReadString(StringBuilder data);
Int32 ReadString(StringBuilder data, Int32 count);
```

ARGUMENTS

Name	Description	Type
`data`	A `StringBuilder` object created by the calling program to hold the string to be read from the formatted read buffer. This method appends the output string to `data`.	`StringBuilder`
`count`	The number of characters to be read from the read buffer.	`Int32`

RETURN VALUE

Type	Description
`String`	The string actually read from the formatted read buffer.
`Int32`	The number of characters actually appended to the `StringBuilder` parameter.

SCANF EQUIVALENTS

The `ReadString` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadString Method	Equivalent Scanf Call
`String ReadString();`	`Scanf("%s", out result);` (return `result`)
`Int32 ReadString(StringBuilder data);`	`Scanf("%s", out data);` `(return data.Length)`
`String ReadString(Int32 count);`	`Scanf("%#s", count, out result);` (return `result`)
`Int32 ReadString(StringBuilder data, Int32 count);`	`Scanf("%#s", count, out data);` (return `data.Length`)

9.4.25. Read

DESCRIPTION

Reads the specified data from the formatted read buffer, and converts it to the type specified by the return type of the method.

DEFINITION

```
Char ReadChar();
Int64 ReadInt64();
UInt64 ReadUInt64();
Double ReadDouble();
```

RETURN VALUE

Type	Description
`Char, Int64, UInt64, Double`	The value read from the formatted read buffer, converted to the specified type.

SCANF EQUIVALENTS

The `Read`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

Read*Type* Method	**Equivalent Scanf Call**
`Char ReadChar();`	`Scanf("%c", out result);` (return `result`)
`UInt64 ReadUInt64();` `Int64 ReadInt64();`	`Scanf("%d", out result);` (return `result`)
`Double ReadDouble();`	`Scanf("%g", out result);` (return `result`)

9.4.26. ReadLine (String)

DESCRIPTION

Reads a string from the formatted read buffer. The read stops when an EOL character is reached.

DEFINITION

```
String ReadLine();
Int32 ReadLine(StringBuilder data);
```

ARGUMENTS

Name	Description	Type
`data`	A `StringBuilder` object created by the calling program to hold the string to be read from the formatted read buffer. This method appends the output string to `data`.	`StringBuilder`

RETURN VALUE

Type	Description
`String`	The string actually read from the formatted read buffer.
`Int32`	The number of characters actually appended to the `StringBuilder` parameter.

SCANF EQUIVALENTS

The `ReadLine` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadLine Method	Equivalent Scanf Call
`String ReadLine();`	`Scanf("%T", out result);` (return `result`)
`Int32 ReadLine(StringBuilder data);`	`Scanf("%T", out data);` (return `data.Length`)

9.4.27. ReadLine

DESCRIPTION

Reads the specified data from the formatted read buffer, reading through the first EOL character, and converts it to the type specified by the return type of the method.

DEFINITION

```
Char ReadLineChar();
Int64 ReadLineInt64();
UInt64 ReadLineUInt64();
Double ReadLineDouble();
```

RETURN VALUE

Type	Description
`Char, Int64, UInt64, Double`	The value read from the formatted read buffer, converted to the specified type.

SCANF EQUIVALENTS

The `ReadLine`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadLine*Type* Method	**Equivalent Scanf Call**
`Char ReadLineChar();`	`Scanf("%c%*T", out result);` (return `result`)
`Int64 ReadLineUInt64();` `Int64 ReadLineInt64();`	`Scanf("%d%*T", out result);` (return `result`)
`Double ReadLineDouble();`	`Scanf("%g%*T", out result);` (return `result`)

9.4.28. ReadList

DESCRIPTION

Reads the specified comma separated list data from the formatted read buffer, and converts it to an array of the type specified by the return type of the method.

DEFINITION

```
Byte[] ReadListOfByte(Int64 count);
Int64 ReadListOfByte(Byte[] data, Int64 index, Int64 count);

SByte[] ReadListOfSByte(Int64 count);
Int64 ReadListOfSByte(SByte[] data, Int64 index, Int64 count);

Int16[] ReadListOfInt16(Int64 count);
Int64 ReadListOfInt16(Int16[] data, Int64 index, Int64 count);

UInt16[] ReadListOfUInt16(Int64 count);
Int64 ReadListOfUInt16(UInt16[] data, Int64 index, Int64 count);

Int32[] ReadListOfInt32(Int64 count);
Int64 ReadListOfInt32(Int32[] data, Int64 index, Int64 count);

UInt32[] ReadListOfUInt32(Int64 count);
Int64 ReadListOfUInt32(UInt32[] data, Int64 index, Int64 count);

Int64[] ReadListOfInt64(Int64 count);
Int64 ReadListOfInt64(Int64[] data, Int64 index, Int64 count);

UInt64[] ReadListOfUInt64(Int64 count);
Int64 ReadListOfUInt64(UInt64[] data, Int64 index, Int64 count);

Single[] ReadListOfSingle(Int64 count);
Int64 ReadListOfSingle(Single[] data, Int64 index, Int64 count);

Double[] ReadListOfDouble(Int64 count);
Int64 ReadListOfDouble(Double[] data, Int64 index, Int64 count);
```

ARGUMENTS

Name	Description	Type
`data`	An array of numbers to be converted from a comma separated list of numbers from the instrument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element of `data` into which values from the list are placed.	`Int64`
`count`	The number of elements from the list to be placed into `data`, starting from `index` (if the overload includes `index`) or the beginning of the array (if the overload does not include `index`).	`Int64`

RETURN VALUE

Type	Description
`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], UInt64[], Single[], Double[]`	The values read from the formatted read buffer, converted to the specified type.
`Int64`	The number of elements in the list actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadListOf`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadListOf*Type* Method	**Equivalent Scanf Call**

`Byte[] ReadListOfByte();`
`UInt16[] ReadListOfUInt16();`
`UInt32[] ReadListOfUInt32();`
`UInt64[] ReadListOfUInt64();`
	`Scanf("%,u", out result);`
	(return `result`)

`SByte[] ReadListOfSByte();`
`Int16[] ReadListOfInt16();`
`Int32[] ReadListOfInt32();`
`Int64[] ReadListOfInt64();`
	`Scanf("%,d", out result);`
	(return `result`)

`Single[] ReadListOfSingle();` `Double[] ReadListOfDouble();`	`Scanf("%,g", out result);` (return `result`)

9.4.29. ReadLineList

DESCRIPTION

Reads the specified comma separated list data from the formatted read buffer, reading through the first EOL character, and converts it to an array of the type specified by the return type of the method.

DEFINITION

```
Byte[] ReadLineListOfByte();
Int64 ReadLineListOfByte(Byte[] data, Int64 index);

SByte[] ReadLineListOfSByte();
Int64 ReadLineListOfSByte(SByte[] data, Int64 index);

Int16[] ReadLineListOfInt16();
Int64 ReadLineListOfInt16(Int16[] data, Int64 index);

UInt16[] ReadLineListOfUInt16();
Int64 ReadLineListOfUInt16(UInt16[] data, Int64 index);

Int32[] ReadLineListOfInt32();
Int64 ReadLineListOfInt32(Int32[] data, Int64 index);

UInt32[] ReadLineListOfUInt32();
Int64 ReadLineListOfUInt32(UInt32[] data, Int64 index);

Int64[] ReadLineListOfInt64();
Int64 ReadLineListOfInt64(Int64[] data, Int64 index);

UInt64[] ReadLineListOfUInt64();
Int64 ReadLineListOfUInt64(UInt64[] data, Int64 index);

Single[] ReadLineListOfSingle();
Int64 ReadLineListOfSingle(Single[] data, Int64 index);

Double[] ReadLineListOfDouble();
Int64 ReadLineListOfDouble(Double[] data, Int64 index);
```

ARGUMENTS

Name	Description	Type
`data`	An array of numbers to be converted from a comma separated list of numbers from the instrument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element of `data` into which values from the list are placed.	`Int64`
`count`	The number of elements from the list to be placed into `data`, starting from `index`.	`Int64`

RETURN VALUE

Type	Description
`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], UInt64[], Single[], Double[]`	The values read from the formatted read buffer, converted to the specified type.
`Int64`	The number of elements in the list actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadLineListOf`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadLineListOf*Type* Method	**Equivalent Scanf Call**

`Byte[] ReadLineListOfByte();`
`UInt16[] ReadLineListOfUInt16();`
`UInt32[] ReadLineListOfUInt32();`
`UInt64[] ReadLineListOfUInt64();`
	`Scanf("%,u%*T", out result);`
	(return `result`)

`SByte[] ReadLineListOfSByte();`
`Int16[] ReadLineListOfInt16();`
`Int32[] ReadLineListOfInt32();`
`Int64[] ReadLineListOfInt64();`
	`Scanf("%,d%*T", out result);`
	(return `result`)

`Single[] ReadLineListOfSingle();` `Double[] ReadLineListOfDouble();`	`Scanf("%,g%*T", out result);` (return `result`)

9.4.30. ReadBinaryBlock

DESCRIPTION

Reads a raw binary array or IEEE_488 definite or indefinite block from the formatted read buffer, and converts it to an array of the type specified by the method name.

DEFINITION

```
Byte[] ReadBinaryBlockOfByte();
Byte[] ReadBinaryBlockOfByte(Boolean seekToBlock);
Int64 ReadBinaryBlockOfByte(Byte[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfByte(Byte[] data, Int64 index, Int64 count,
                            Boolean seekToBlock);

SByte[] ReadBinaryBlockOfSByte();
SByte[] ReadBinaryBlockOfSByte(Boolean seekToBlock);
Int64 ReadBinaryBlockOfSByte(SByte[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfSByte(SByte[] data, Int64 index, Int64 count,
                             Boolean seekToBlock);

Int16[] ReadBinaryBlockOfInt16();
Int16[] ReadBinaryBlockOfInt16(Boolean seekToBlock);
Int64 ReadBinaryBlockOfInt16(Int16[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfInt16(Int16[] data, Int64 index, Int64 count,
                             Boolean seekToBlock);

UInt16[] ReadBinaryBlockOfUInt16();
UInt16[] ReadBinaryBlockOfUInt16(Boolean seekToBlock);
Int64 ReadBinaryBlockOfUInt16(UInt16[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfUInt16(UInt16[] data, Int64 index, Int64 count,
                              Boolean seekToBlock);

Int32[] ReadBinaryBlockOfInt32();
Int32[] ReadBinaryBlockOfInt32(Boolean seekToBlock);
Int64 ReadBinaryBlockOfInt32(Int32[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfInt32(Int32[] data, Int64 index, Int64 count,
                             Boolean seekToBlock);

UInt32[] ReadBinaryBlockOfUInt32();
UInt32[] ReadBinaryBlockOfUInt32(Boolean seekToBlock);
Int64 ReadBinaryBlockOfUInt32(UInt32[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfUInt32(UInt32[] data, Int64 index, Int64 count,
                              Boolean seekToBlock);

Int64[] ReadBinaryBlockOfInt64();
Int64 ReadBinaryBlockOfInt64(Int64[] data, Int64 index, Int64 count);

UInt64[] ReadBinaryBlockOfUInt64();
Int64 ReadBinaryBlockOfUInt64(UInt64[] data, Int64 index, Int64 count);

Single[] ReadBinaryBlockOfSingle();
Single[] ReadBinaryBlockOfSingle(Boolean seekToBlock);
```

```
Int64 ReadBinaryBlockOfSingle(Single[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfSingle(Single[] data, Int64 index, Int64 count,
                              Boolean seekToBlock);

Double[] ReadBinaryBlockOfDouble();
Double[] ReadBinaryBlockOfDouble(Boolean seekToBlock);
Int64 ReadBinaryBlockOfDouble(Double[] data, Int64 index, Int64 count);
Int64 ReadBinaryBlockOfDouble(Double[] data, Int64 index, Int64 count,
                              Boolean seekToBlock);
```

ARGUMENTS

Name	Description	Type
`data`	An array of numbers be converted from the raw binary array or IEEE-488 block of numbers from the instrument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element of `data` into which values from the block are placed.	`Int64`
`count`	The number of elements from the block to be placed into `data`, starting from `index`.	`Int64`
`seekToBlock`	If `true`, read and discard characters that preceed the IEEE-488 block. If `false`, the first character read must be the start of the block. This argument is only used if the `BinaryEncoding` is `DefiniteLengthBlockData` or `IndefiniteLengthBlockData` The default value is `false`.	`Boolean`

RETURN VALUE

Type	Description
`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], UInt64[], Single[], Double[]`	The values read from the formatted read buffer, converted to the specified type.
`Int64`	The number of elements in the list actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadBinaryBlockOf`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadBinaryBlockOfOf*Type* Method	**Equivalent Scanf Call**
If BinaryEncoding = DefiniteLengthBlockData	
`Byte[] ReadBinaryBlockOfByte();` `SByte[] ReadBinaryBlockOfSByte();`	`Scanf("%b", out result)` (return `result`)
`Int16[] ReadBinaryBlockOfInt16();` `UInt16[] ReadBinaryBlockOfUInt16();`	`Scanf("%hb", out result)` (return `result`)
`Int32[] ReadBinaryBlockOfInt32();` `UInt32[] ReadBinaryBlockOfUInt32();`	`Scanf("%lb", out result)` (return `result`)

`Int64[] ReadBinaryBlockOfInt64();` `UInt64[] ReadBinaryBlockOfUInt64();`	`ReadBinaryBlockOfInt64()` and `ReadBinaryBlockOfUInt64()` for definite length blocks and the corresponding Scanf format specifier (`%llb`) are not supported at this time.
`Single[] ReadBinaryBlockOfSingle();`	`Scanf("%zb", out result)` (return `result`)
`Double[] ReadBinaryBlockOfDouble();`	`Scanf("%Zb", out result)` (return `result`)
If BinaryEncoding = IndefiniteLengthBlockData	
`Byte[] ReadBinaryBlockOfByte();` `SByte[] ReadBinaryBlockOfSByte();`	`Scanf("%b", out result)` (return `result`)
`Int16[] ReadBinaryBlockOfInt16();` `UInt16[] ReadBinaryBlockOfUInt16();`	`Scanf("%hb", out result)` (return `result`)
`Int32[] ReadBinaryBlockOfInt32();` `UInt32[] ReadBinaryBlockOfUInt32();`	`Scanf("%lb", out result)` (return `result`)
`Int64[] ReadBinaryBlockOfInt64();` `UInt64[] ReadBinaryBlockOfUInt64();`	`ReadBinaryBlockOfInt64()` and `ReadBinaryBlockOfUInt64()` for indefinite length blocks and the corresponding Scanf format specifier (`%llb`) are not supported at this time.
`Single[] ReadBinaryBlockOfSingle();`	`Scanf("%zb", out result)` (return `result`)
`Double[] ReadBinaryBlockOfDouble();`	`Scanf("%Zb", out result)` (return `result`)
If BinaryEncoding = RawBigEndian	
`Byte[] ReadBinaryBlockOfByte();` `SByte[] ReadBinaryBlockOfSByte();`	`Scanf("%!oby", out result)` (return `result`)
`Int16[] ReadBinaryBlockOfInt16();` `UInt16[] ReadBinaryBlockOfUInt16();`	`Scanf("%!obhy", out result)` (return `result`)
`Int32[] ReadBinaryBlockOfInt32();` `UInt32[] ReadBinaryBlockOfUInt32();`	`Scanf("%!obly", out result)` (return `result`)
`Int64[] ReadBinaryBlockOfInt64();` `UInt64[] ReadBinaryBlockOfUInt64();`	`Scanf("%!oblly", out result)` (return `result`)
`Single[] ReadBinaryBlockOfSingle();`	`ReadBinaryBlockOfSingle()` for raw big endian arrays and the corresponding Scanf format specifier (%!obzy) are not supported at this time.
`Double[] ReadBinaryBlockOfDouble();`	`ReadBinaryBlockOfDouble()` for raw big endian arrays and the corresponding Scanf format specifier (%!obZy) are not supported at this time.
If BinaryEncoding = RawLittleEndian	
`Byte[] ReadBinaryBlockOfByte();` `SByte[] ReadBinaryBlockOfSByte();`	`Scanf("%!oly", out result)` (return `result`)

`Int16[] ReadBinaryBlockOfInt16();` `UInt16[] ReadBinaryBlockOfUInt16();`	`Scanf("%!olhy", out result)` (return `result`)
`Int32[] ReadBinaryBlockOfInt32();` `UInt32[] ReadBinaryBlockOfUInt32();`	`Scanf("%!olly", out result)` (return `result`)
`Int64[] ReadBinaryBlockOfInt64();` `UInt64[] ReadBinaryBlockOfUInt64();`	`Scanf("%!ollly", out result)` (return `result`)
`Single[] ReadBinaryBlockOfSingle();`	`ReadBinaryBlockOfSingle()` for raw little endian arrays and the corresponding Scanf format specifier (%!olzy) are not supported at this time.
`Double[] ReadBinaryBlockOfDouble();`	`ReadBinaryBlockOfDouble()` for raw little endian arrays and the corresponding Scanf format specifier (%!olZy) are not supported at this time.

The way that the binary data is treated is not affected by the value of the `seekToBlock` parameter.

9.4.31. ReadLineBinaryBlock

DESCRIPTION

Reads a raw binary array or IEEE_488 definite or indefinite block from the formatted read buffer, reading through the first EOL character, and converts it to an array of the type specified by the method name.

DEFINITION

```
Byte[] ReadLineBinaryBlockOfByte();
Byte[] ReadLineBinaryBlockOfByte(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfByte(Byte[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfByte(Byte[] data, Int64 index, Int64 count,
                                Boolean seekToBlock);

SByte[] ReadLineBinaryBlockOfSByte();
SByte[] ReadLineBinaryBlockOfSByte(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfSByte(SByte[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfSByte(SByte[] data, Int64 index, Int64 count,
                                 Boolean seekToBlock);

Int16[] ReadLineBinaryBlockOfInt16();
Int16[] ReadLineBinaryBlockOfInt16(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfInt16(Int16[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfInt16(Int16[] data, Int64 index, Int64 count,
                                 Boolean seekToBlock);

UInt16[] ReadLineBinaryBlockOfUInt16();
UInt16[] ReadLineBinaryBlockOfUInt16(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfUInt16(UInt16[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfUInt16(UInt16[] data, Int64 index, Int64 count,
                                  Boolean seekToBlock);

Int32[] ReadLineBinaryBlockOfInt32();
Int32[] ReadLineBinaryBlockOfInt32(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfInt32(Int32[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfInt32(Int32[] data, Int64 index, Int64 count,
                                 Boolean seekToBlock);

UInt32[] ReadLineBinaryBlockOfUInt32();
UInt32[] ReadLineBinaryBlockOfUInt32(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfUInt32(UInt32[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfUInt32(UInt32[] data, Int64 index, Int64 count,
                                  Boolean seekToBlock);

Int64[] ReadLineBinaryBlockOfInt64();
Int64 ReadLineBinaryBlockOfInt64(Int64[] data, Int64 index, Int64 count);

UInt64[] ReadLineBinaryBlockOfUInt64();
Int64 ReadLineBinaryBlockOfUInt64(UInt64[] data, Int64 index, Int64 count);

Single[] ReadLineBinaryBlockOfSingle();
Single[] ReadLineBinaryBlockOfSingle(Boolean seekToBlock);
```

```
Int64 ReadLineBinaryBlockOfSingle(Single[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfSingle(Single[] data, Int64 index, Int64 count,
                                  Boolean seekToBlock);

Double[] ReadLineBinaryBlockOfDouble();
Double[] ReadLineBinaryBlockOfDouble(Boolean seekToBlock);
Int64 ReadLineBinaryBlockOfDouble(Double[] data, Int64 index, Int64 count);
Int64 ReadLineBinaryBlockOfDouble(Double[] data, Int64 index, Int64 count,
                                  Boolean seekToBlock);
```

ARGUMENTS

Name	Description	Type
`data`	An array of numbers be converted from the raw binary array or IEEE-488 block of numbers from the instrument.	`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], Uint64[], Single[], Double[]`
`index`	The index of the first element of `data` into which values from the block are placed.	`Int64`
`count`	The number of elements from the block to be placed into `data`, starting from `index`.	`Int64`
`seekToBlock`	If `true`, read and discard characters that preceed the IEEE-488 block. If `false`, the first character read must be the start of the block. This argument is only used if the `BinaryEncoding` is `DefiniteLengthBlockData` or `IndefiniteLengthBlockData` The default value is `false`.	`Boolean`

RETURN VALUE

Type	Description
`Byte[], SByte[], Int16[], UInt16[], Int32[], UInt32[], Int64[], UInt64[], Single[], Double[]`	The values read from the formatted read buffer, converted to the specified type.
`Int64`	The number of elements in the list actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadLineBinaryBlockOf`*`Type`* method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadLineBinaryBlockOfOf*Type* Method	**Equivalent Scanf Call**
If BinaryEncoding = DefiniteLengthBlockData	
`Byte[] ReadLineBinaryBlockOfByte();` `SByte[] ReadLineBinaryBlockOfSByte();`	`Scanf("%b%*T", out result)` (return `result`)
`Int16[] ReadLineBinaryBlockOfInt16();` `UInt16[] ReadLineBinaryBlockOfUInt16();`	`Scanf("%hb%*T", out result)` (return `result`)
`Int32[] ReadLineBinaryBlockOfInt32();` `UInt32[] ReadLineBinaryBlockOfUInt32();`	`Scanf("%lb%*T", out result)` (return `result`)

`Int64[] ReadLineBinaryBlockOfInt64();` `UInt64[] ReadLineBinaryBlockOfUInt64();`	`ReadLineBinaryBlockOfInt64()` and `ReadLineBinaryBlockOfUInt64()` for definite length blocks and the corresponding Scanf format specifier (`%llb`) are not supported at this time.
`Single[] ReadLineBinaryBlockOfSingle();`	`Scanf("%zb%*T", out result)` (return `result`)
`Double[] ReadLineBinaryBlockOfDouble();`	`Scanf("%Zb%*T", out result)` (return `result`)
If BinaryEncoding = IndefiniteLengthBlockData	
`Byte[] ReadLineBinaryBlockOfByte();` `SByte[] ReadLineBinaryBlockOfSByte();`	`Scanf("%b%*T", out result)` (return `result`)
`Int16[] ReadLineBinaryBlockOfInt16();` `UInt16[] ReadLineBinaryBlockOfUInt16();`	`Scanf("%hb%*T", out result)` (return `result`)
`Int32[] ReadLineBinaryBlockOfInt32();` `UInt32[] ReadLineBinaryBlockOfUInt32();`	`Scanf("%lb%*T", out result)` (return `result`)
`Int64[] ReadLineBinaryBlockOfInt64();` `UInt64[] ReadLineBinaryBlockOfUInt64();`	`ReadLineBinaryBlockOfInt64()` and `ReadLineBinaryBlockOfUInt64()` for indefinite length blocks and the corresponding Scanf format specifier (`%llb`) are not supported at this time.
`Single[] ReadLineBinaryBlockOfSingle();`	`Scanf("%zb%*T", out result)` (return `result`)
`Double[] ReadLineBinaryBlockOfDouble();`	`Scanf("%Zb%*T", out result)` (return `result`)
If BinaryEncoding = RawBigEndian	
`Byte[] ReadLineBinaryBlockOfByte();` `SByte[] ReadLineBinaryBlockOfSByte();`	`Scanf("%y%*T", out result)` (return `result`)
`Int16[] ReadLineBinaryBlockOfInt16();` `UInt16[] ReadLineBinaryBlockOfUInt16();`	`Scanf("%hy%*T", out result)` (return `result`)

`Int32[] ReadLineBinaryBlockOfInt32();`
`UInt32[] ReadLineBinaryBlockOfUInt32();`
`Single[] ReadLineBinaryBlockOfSingle();`
	`Scanf("%ly%*T", out result)`
	(return `result`)

`Int64[] ReadLineBinaryBlockOfInt64();`
`UInt64[] ReadLineBinaryBlockOfUInt64();`
`Double[] ReadLineBinaryBlockOfDouble();`
	`Scanf("%lly%*T", out result)`
	(return `result`)

`Single[] ReadLineBinaryBlockOfSingle();`	`ReadLineBinaryBlockOfSingle()` for raw big endian arrays and the corresponding Scanf format specifier (%!obzy) are not supported at this time.
`Double[] ReadLineBinaryBlockOfDouble();`	`ReadLineBinaryBlockOfDouble()` for raw big endian arrays and the corresponding Scanf format specifier (%!obZy) are not supported at this time.
If BinaryEncoding = RawLittleEndian	

`Byte[] ReadLineBinaryBlockOfByte();` `SByte[] ReadLineBinaryBlockOfSByte();`	`Scanf("%!oly%*T", out result)` (return `result`)
`Int16[] ReadLineBinaryBlockOfInt16();` `UInt16[] ReadLineBinaryBlockOfUInt16();`	`Scanf("%!olhy%*T", out result)` (return `result`)

`Int32[] ReadLineBinaryBlockOfInt32();`
`UInt32[] ReadLineBinaryBlockOfUInt32();`
`Single[] ReadLineBinaryBlockOfSingle();`
	`Scanf("%!olly%*T", out result)`
	(return `result`)

`Int64[] ReadLineBinaryBlockOfInt32();`
`UInt64[] ReadLineBinaryBlockOfUInt32();`
`Double[] ReadLineBinaryBlockOfDouble();`
	`Scanf("%!ollly%*T", out result)`
	(return `result`)

`Single[] ReadLineBinaryBlockOfSingle();`	`ReadLineBinaryBlockOfSingle()` for raw little endian arrays and the corresponding Scanf format specifier (%!olzy) are not supported at this time.
`Double[] ReadLineBinaryBlockOfDouble();`	`ReadLineBinaryBlockOfDouble()` for raw little endian arrays and the corresponding Scanf format specifier (%!olZy) are not supported at this time.

The way that the binary data is treated is not affected by the value of the `seekToBlock` parameter.

9.4.32. ReadWhileMatch

DESCRIPTION

Reads an arbitrary number of characters that match a specified list of characters. The method stops reading at the first non-matching character, which remains in the read buffer. There is no processing of ranges or other meta-characters. The method will read additional characters from the instrument to perform this operation, if necessary.

DEFINITION

```
String ReadWhileMatch(String characters);
```

ARGUMENTS

Name	Description	Type
`characters`	A string of literal characters to be matched as individual characters.	`String`

RETURN VALUE

Type	Description
`String`	The string actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadWhileMatch` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table. There is no processing of ranges or other meta-characters associated with the "[]" flag in Scanf.

ReadWhileMatch Method	Equivalent Scanf Call
`String ReadWhileMatch(String characters);`	`Scanf("%[<characters>]", out result);` (return `result`)

9.4.33. ReadUntilMatch

DESCRIPTION

Reads an arbitrary number of characters until a matching character is found. The method stops reading at the first matching character, which is discarded from the read buffer unless the `discardMatch` parameter is specified and is `false`. The string returned does not include the matched character. There is no processing of ranges or other meta-characters. The method will read additional characters from the instrument to perform this operation, if necessary.

DEFINITION

```
String ReadUntilMatch(Char ch);
String ReadUntilMatch(String characters, Boolean discardMatch);
```

ARGUMENTS

Name	Description	Type
`ch`	A single literal character to be matched.	`String`
`characters`	A string of literal characters to be matched as individual characters.	`String`
`discardMatch`	If `true`, the first character in the read buffer that matches a character in `characters` is consumed and discarded. If `false`, the matched character remains in the formatted I/O buffer. The default value is `true`.	`Boolean`

RETURN VALUE

Type	Description
`String`	The string actually read from the formatted read buffer. This string does not include the matched character.

SCANF EQUIVALENTS

The `ReadUntilMatch` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table. There is no processing of ranges or other meta-characters associated with the "[]" flag in Scanf.

ReadUntilMatch Method	Equivalent Scanf Call
`String ReadUntilMatch(Char ch);`	`Scanf("%[^<ch>]", out result);` (return `result`)
`String ReadUntilMatch(String characters, Boolean discardMatch);`	`Scanf("%[^<characters>]", out result);` (return `result`)

9.4.34. ReadUntilEnd

DESCRIPTION

Reads an arbitrary number of characters until an END or termination character is found. The method stops reading at the first END or termination character. The string returned includes the character with the END indicator or termination character.

If the underlying protocol does not support END or the termination character, this method may time out or exhibit other implementation-specific behavior.

DEFINITION

```
String ReadUntilEnd();
```

RETURN VALUE

Type	Description
`String`	The string actually read from the formatted read buffer.

SCANF EQUIVALENTS

The `ReadUntilEnd` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table.

ReadUntilEnd Method	Equivalent Scanf Call
`String ReadUntilEnd();`	`Scanf("%t", out result);` (return `result`)

9.4.35. Introduction to Formatted Skip Methods

Formatted skip methods include `Skip, SkipString, and SkipUntilEnd` in the `IMessageBasedFormattedIO` interface.

Skip methods differ from Read methods in that Skip methods do not return the data skipped.

The section that describes each Skip method also includes the equivalent Scanf format specifier. To determine the corresponding behavior, refer to sections 9.4.13.3, Scanf Format Specifier Usage Summary, and 9.4.14, Scanf, for details.

9.4.36. Skip

DESCRIPTION

Reads and removes up to `count` characters from the formatted read buffer. The method will read additional characters from the instrument to perform this operation, if necessary, but will not skip over an END.

DEFINITION

```
void Skip(Int64 count);
```

ARGUMENTS

Name	Description	Type
`count`	The number of characters to remove from the buffer.	`Int64`

SCANF EQUIVALENTS

The `Skip` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table, where the data read is discarded as indicated by the '*' flag.

Skip Method	Equivalent Scanf Call
`void Skip(Int64 count);`	`Scanf("%*<count>c");`

9.4.37. SkipString

DESCRIPTION

Skip and discard the exact string specified by `data` from the formatted I/O read buffer. Multiple whitespace characters in the read buffer may match a single whitespace in the data string. If data contains a % character, the method throws an exception. The method will read additional characters from the instrument to perform this operation, if necessary, but will not skip over an END.

This method will throw an exception if the data read is not a match for the specified data.

DEFINITION

```
void SkipString (String data);
```

ARGUMENTS

Name	Description	Type
`data`	The string to be read from the formatted read buffer. The string may not contain the '%' character.	`String`

SCANF EQUIVALENTS

The `SkipString` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table, where the data read (in the `dataRead` parameter) is discarded.

SkipString Method	Equivalent Scanf Call
`void SkipString (String data);`	`Scanf(data, out dataRead);`

9.4.38. SkipUntilEnd

DESCRIPTION

Discards the entire formatted I/O read buffer. If the previous formatted I/O buffer did not include an END or termination character, this method reads from the device until an END or termination character is encountered and discards the data.

DEFINITION

```
void SkipUntilEnd();
```

SCANF EQUIVALENTS

The `SkipUntilEnd` method implementations exhibit exactly the same behavior (but not necessarily implemented exactly as shown) as a corresponding call to `Scanf`, as shown in the following table, where the data read is discarded as indicated by the '*' flag.

SkipUntilEnd Method	Equivalent Scanf Call
`void SkipUntilEnd();`	`Scanf("%*t");`

9.5. FormattedIO Implementations

The IVI Foundation provides a standard implementation of the `IMessageBasedFormattedIO` interface. The implementation is the `MessageBasedFormattedIO` class.

RECOMMENDATION 9.5.1

The recommendation is that VISA.NET vendors use the standard IVI `MessageBasedFormattedIO` class for formatted I/O.

OBSERVATION 9.5.1

The `MessageBasedFormattedIO` class is public, and the IVI Foundation cannot prevent arbitrary clients from using it. However, it is intended for use by VISA.NET vendors only. Because it is intended for use by VISA.NET vendors only, there are no IntelliSense comments that describe the class API.

9.5.2. MessageBasedFormattedIO Constructors

DESCRIPTION

Create an instance of the `MessageBasedFormattedIO` class.

DEFINITION

```
public MessageBasedFormattedIO(IMessageBasedSession session)
```

ARGUMENTS

Name	Description	Type
`session`	A reference to the message based session to be used by formatted I/O to perform lower level I/O operations.	`IMessageBasedSession`

IMPLEMENTATION

OBSERVATION 9.5.2

The `session` parameter must reference a complete implementation of `IMessageBasedSession`, as this is required for the proper operation of the `MessageBasedFormattedIO` class.

Section 10: Register Based Session Interfaces

Register based resources are controlled by accessing device registers or memory, or both. Depending on the resource type, they may also support message based operations. Refer to VPP4.3 section 5.1 for more information about register based resources. The functionality of INSTR resources is broken up into several interfaces in VISA.NET I/O.

10.1. IRegisterBasedSession

DESCRIPTION

The base session type for register-based devices.

DEFINITION

```
public interface IRegisterBasedSession : IVisaSession
{

   Boolean AllowDma { get; set; }
   Int32 DestinationIncrement { get; set; }
   Int32 SourceIncrement { get; set; }

   IMemoryMap MapAddress(AddressSpace space, Int64 offset, Int64 size);

   Byte In8(AddressSpace space, Int64 sourceOffset);
   Int16 In16(AddressSpace space, Int64 sourceOffset);
   Int32 In32(AddressSpace space, Int64 sourceOffset);
   Int64 In64(AddressSpace space, Int64 sourceOffset);

   void Out8(AddressSpace space, Int64 destinationOffset, Byte value);
   void Out16(AddressSpace space, Int64 destinationOffset, Int16 value);
   void Out32(AddressSpace space, Int64 destinationOffset, Int32 value);
   void Out64(AddressSpace space, Int64 destinationOffset, Int64 value);

   Byte[] MoveIn8(AddressSpace space, Int64 sourceOffset, Int64 count);
   void MoveIn8(AddressSpace space, Int64 sourceOffset, Int64 count,
                Byte[] destinationBuffer, Int64 destinationIndex);

   Int16[] MoveIn16(AddressSpace space, Int64 sourceOffset, Int64 count);
   void MoveIn16(AddressSpace space, Int64 sourceOffset, Int64 count,
                 Int16[] destinationBuffer, Int64 destinationIndex);

   Int32[] MoveIn32(AddressSpace space, Int64 sourceOffset, Int64 count);
   void MoveIn32(AddressSpace space, Int64 sourceOffset, Int64 count,
                 Int32[] destinationBuffer, Int64 destinationIndex);

   Int64[] MoveIn64(AddressSpace space, Int64 sourceOffset, Int64 count);
   void MoveIn64(AddressSpace space, Int64 sourceOffset, Int64 count,
                 Int64[] destinationBuffer, Int64 destinationIndex);

   void MoveOut8(AddressSpace space, Int64 destinationOffset,
                 Byte[] sourceBuffer);
   void MoveOut8(AddressSpace space, Int64 destinationOffset,
                 Byte[] sourceBuffer, Int64 sourceIndex, Int64 count);

   void MoveOut16(AddressSpace space, Int64 destinationOffset,
                  Int16[] sourceBuffer);
   void MoveOut16(AddressSpace space, Int64 destinationOffset,
                  Int16[] sourceBuffer, Int64 sourceIndex, Int64 count);
```

```
    void MoveOut32(AddressSpace space, Int64 destinationOffset,
                   Int32[] sourceBuffer);
    void MoveOut32(AddressSpace space, Int64 destinationOffset,
                   Int32[] sourceBuffer, Int64 sourceIndex, Int64 count);

    void MoveOut64(AddressSpace space, Int64 destinationOffset,
                   Int64[] sourceBuffer);
    void MoveOut64(AddressSpace space, Int64 destinationOffset,
                   Int64[] sourceBuffer, Int64 sourceIndex, Int64 count);
}
```

CORRESPONDING VISA FEATURES

The `IRegisterBasedSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IRegisterBasedSession`.

Property Name	**VISA Attribute Name**
`AllowDma`	`VI_ATTR_DMA_ALLOW_EN`
`DestinationIncrement`	`VI_ATTR_DEST_INCREMENT`
`SourceIncrement`	`VI_ATTR_SRC_INCREMENT`

The `IRegisterBasedSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IRegisterBasedSession`.

Method Name	**VISA Function Name**
`MapAddress`	`viMapAddressEx`
`In8`	`viIn8Ex`
`In16`	`viIn16Ex`
`In32`	`viIn32Ex`
`In64`	`viIn64Ex`
`Out8`	`viOut8Ex`
`Out16`	`viOut16Ex`
`Out32`	`viOut32Ex`
`Out64`	`viOut64Ex`
`MoveIn8`	`viMoveIn8Ex`
`MoveIn16`	`viMoveIn16Ex`
`MoveIn32`	`viMoveIn32Ex`
`MoveIn64`	`viMoveIn64Ex`
`MoveOut8`	`viMoveOut8Ex`
`MoveOut16`	`viMoveOut16Ex`
`MoveOut32`	`viMoveOut32Ex`
`MoveOut64`	`viMoveOut64Ex`

IMPLEMENTATION

RULE 10.1.1

VISA.NET I/O register based session classes **SHALL** implement `IRegisterBasedSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 10.1.2

All VISA.NET I/O session classes that implement the VXI and GPIB-VXI INSTR resources **SHALL** implement the `IRegisterBasedSession` interface.

10.2. IMemoryMap

DESCRIPTION

Provides memory mapping services for register-based devices.

DEFINITION

```
public interface IMemoryMap : IDisposable
{
   AddressSpace AddressSpace { get; }
   Int64 BaseAddress { get; }
   Int64 Size { get; }
   IntPtr VirtualAddress { get; }

   Byte Peek8(Int64 offset);
   Int16 Peek16(Int64 offset);
   Int32 Peek32(Int64 offset);
   Int64 Peek64(Int64 offset);

   void Poke8(Int64 offset, Byte value);
   void Poke16(Int64 offset, Int16 value);
   void Poke32(Int64 offset, Int32 value);
   void Poke64(Int64 offset, Int64 value);
}
```

CORRESPONDING VISA FEATURES

The `IMemoryMap` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute correspondence for `IMemoryMap`.

Property Name	VISA Attribute Name
AddressSpace	N/A (this is the parameter passed to MapAddress.)
BaseAddress	VI_ATTR_WIN_BASE_ADDR_64
Size	VI_ATTR_WIN_SIZE_64
VirtualAddress	N/A (this is the output pointer from viMapAddressEx if the pointer can be dereferenced; otherwise, this is IntPtr.Zero)

The `IMemoryMap` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IMemoryMap`.

Method Name	VISA Function Name
Peek8	viPeek8
Peek16	viPeek16
Peek32	viPeek32
Peek64	viPeek64
Poke8	viPoke8
Poke16	viPoke16
Poke32	viPoke32
Poke64	viPoke64

`Dispose`	`viUnmapAddress`

Section 11: INSTR Resources

The INSTR session type lets a controller interact with the device associated with this session type, 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.

11.1. IGpibSession

DESCRIPTION

The INSTR session type for GPIB devices.

DEFINITION

```
public interface IGpibSession : IMessageBasedSession
{
   Boolean AllowDma { get; set; }
   Int16 PrimaryAddress { get; }
   Boolean ReaddressingEnabled { get; set; }
   LineState RenState { get; }

   Int16 SecondaryAddress { get; }
   Boolean UnaddressingEnabled { get; set; }

   void SendRemoteLocalCommand(RemoteLocalMode mode);
   void SendRemoteLocalCommand(GpibInstrumentRemoteLocalMode mode);
}
```

CORRESPONDING VISA FEATURES

The `IGpibSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IGpibSession`.

Property Name	VISA Attribute Name
`AllowDma`	`VI_ATTR_DMA_ALLOW_EN`
`PrimaryAddress`	`VI_ATTR_GPIB_PRIMARY_ADDR`
`ReaddressingEnabled`	`VI_ATTR_GPIB_READDR_EN`
`RenState`	`VI_ATTR_GPIB_REN_STATE`
`SecondaryAddress`	`VI_ATTR_GPIB_SECONDARY_ADDR`
`UnaddressingEnabled`	`VI_ATTR_GPIB_UNADDR_EN`

The `IGpibSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IGpibSession`.

Method Name	VISA Function Name
`SendRemoteLocalCommand`	viGpibControlREN

IMPLEMENTATION

RULE 11.1.1

VISA.NET I/O GPIB and GPIB_VXI INSTR session classes **SHALL** implement `IGpibSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.1.2

All VISA.NET I/O session classes that implement the GPIB and GPIB-VXI INSTR resources **SHALL** implement the interface `IGpibSession`.

11.2. IPxiSession

DESCRIPTION

The INSTR session type for PXI devices.

DEFINITION

```
public interface IPxiSession : IRegisterBasedSession
{
   event EventHandler<PxiInterruptEventArgs> Interrupt;

   Int16 ActualLinkWidth { get; }
   Boolean AllowWriteCombining { get; set; }
   Int16 BusNumber { get; }
   Int16 ChassisNumber { get; }
   Int16 DeviceNumber { get; }
   Int16 DstarBusNumber { get; }
   Int16 DstarLineSet { get; }
   Int16 FunctionNumber { get; }
   Boolean IsExpress { get; }
   Int16 ManufacturerId { get; }
   String ManufacturerName { get; }
   Int16 MaxLinkWidth { get; }
   PxiMemoryType MemTypeBar0 { get; }
   PxiMemoryType MemTypeBar1 { get; }
   PxiMemoryType MemTypeBar2 { get; }
   PxiMemoryType MemTypeBar3 { get; }
   PxiMemoryType MemTypeBar4 { get; }
   PxiMemoryType MemTypeBar5 { get; }
   Int64 MemBaseBar0 { get; }
   Int64 MemBaseBar1 { get; }
   Int64 MemBaseBar2 { get; }
   Int64 MemBaseBar3 { get; }
   Int64 MemBaseBar4 { get; }
   Int64 MemBaseBar5 { get; }
   Int64 MemSizeBar0 { get; }
   Int64 MemSizeBar1 { get; }
   Int64 MemSizeBar2 { get; }
   Int64 MemSizeBar3 { get; }
   Int64 MemSizeBar4 { get; }
   Int64 MemSizeBar5 { get; }
   Int16 ModelCode { get; }
   String ModelName { get; }
   Int16 Slot { get; }
   Int16 SlotLinkWidth { get; }
   Int16 SlotLocalBusLeft { get; }
   Int16 SlotLocalBusRight { get; }
   String SlotPath { get; }   Int16 StarTriggerBus { get; }
   Int16 StarTriggerLine { get; }
   Int16 TriggerBus { get; }

#if NET6_0_OR_GREATER
```

```
   Int16 SlotOffset { get; }
   Int16 SlotWidth { get; }
#endif

   void ReserveTrigger(TriggerLine line);
   void UnreserveTrigger(TriggerLine line);
}
```

CORRESPONDING VISA FEATURES

The `IPxiSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IPxiSession`.

Property Name	**VISA Attribute Name**
ActualLinkWidth	VI_ATTR_PXI_ACTUAL_LWIDTH
AllowWriteCombining	VI_ATTR_PXI_ALLOW_WRITE_COMBINE
BusNumber	VI_ATTR_PXI_BUS_NUM
ChassisNumber	VI_ATTR_PXI_CHASSIS
DeviceNumber	VI_ATTR_PXI_DEV_NUM
DstarBusNumber	VI_ATTR_PXI_DSTAR_BUS
DstarLineSet	VI_ATTR_PXI_DSTAR_SET
FunctionNumber	VI_ATTR_PXI_FUNC_NUM
IsExpress	VI_ATTR_PXI_IS_EXPRESS
ManufacturerId	VI_ATTR_MANF_ID
ManufacturerName	VI_ATTR_MANF_NAME
MaxLinkWidth	VI_ATTR_PXI_MAX_LWIDTH
MemTypeBar0	VI_ATTR_PXI_MEM_TYPE_BAR0
MemTypeBar1	VI_ATTR_PXI_MEM_TYPE_BAR1
MemTypeBar2	VI_ATTR_PXI_MEM_TYPE_BAR2
MemTypeBar3	VI_ATTR_PXI_MEM_TYPE_BAR3
MemTypeBar4	VI_ATTR_PXI_MEM_TYPE_BAR4
MemTypeBar5	VI_ATTR_PXI_MEM_TYPE_BAR5
MemBaseBar0	VI_ATTR_PXI_MEM_BASE_BAR0
MemBaseBar1	VI_ATTR_PXI_MEM_BASE_BAR1
MemBaseBar2	VI_ATTR_PXI_MEM_BASE_BAR2
MemBaseBar3	VI_ATTR_PXI_MEM_BASE_BAR3
MemBaseBar4	VI_ATTR_PXI_MEM_BASE_BAR4
MemBaseBar5	VI_ATTR_PXI_MEM_BASE_BAR5
MemSizeBar0	VI_ATTR_PXI_MEM_SIZE_BAR0
MemSizeBar1	VI_ATTR_PXI_MEM_SIZE_BAR1
MemSizeBar2	VI_ATTR_PXI_MEM_SIZE_BAR2
MemSizeBar3	VI_ATTR_PXI_MEM_SIZE_BAR3
MemSizeBar4	VI_ATTR_PXI_MEM_SIZE_BAR4
MemSizeBar5	VI_ATTR_PXI_MEM_SIZE_BAR5
ModelCode	VI_ATTR_MODEL_CODE
ModelName	VI_ATTR_MODEL_NAME
Slot	VI_ATTR_SLOT

SlotLinkWidth	VI_ATTR_PXI_SLOT_LWIDTH
SlotLocalBusLeft	VI_ATTR_PXI_SLOT_LBUS_LEFT
SlotLocalBusRight	VI_ATTR_PXI_SLOT_LBUS_RIGHT
SlotPath	VI_ATTR_PXI_SLOTPATH
StarTriggerBus	VI_ATTR_PXI_STAR_TRIG_BUS
StarTriggerLine	VI_ATTR_PXI_STAR_TRIG_LINE
TriggerBus	VI_ATTR_PXI_TRIG_BUS
SlotOffset	VI_ATTR_PXI_SLOT_WIDTH
SlotWidth	VI_ATTR_PXI_SLOT_OFFSET

The IPxiSession interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for IPxiSession.

Method Name	VISA Function Name
ReserveTrigger	viAssertTrigger with VI_TRIG_PROT_RESERVE
UnreserveTrigger	viAssertTrigger with VI_TRIG_PROT_UNRESERVE
N/A	viMoveEx[1] viMoveAsyncEx

The IPxiSession interface has one .NET event that correspond to an event defined in VISA. The following table shows correspondence for IPxiSession.

Event Name	VISA Function Name
Interrupt	VI_EVENT_PXI_INTR

IMPLEMENTATION

OBSERVATION 11.2.1

SlotOffset and SlotWidth are omitted from the .NET Framework API for this interface. In the .NET Framework API, they are located in IPxiSession2.

RULE 11.2.2

VISA.NET I/O PXI session classes **SHALL** implement IPxiSession interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.2.3

All VISA.NET I/O session classes that implement the PXI resources **SHALL** implement the interface IPxiSession.

11.2.2. IPxiSession2 (.NET Framework Only)

DESCRIPTION

In the .NET Framework API, the INSTR session type for PXI devices. This derives from and supercedes IPxiSession for the .NET Framework API only.

DEFINITION

[1] Refer to the footnote in Section 12.2.

```
#if NETFRAMEWORK
public interface IPxiSession2 : IPxiSession
{
   Int16 SlotWidth { get; }
   Int16 SlotOffset { get; }
}
#endif
```

CORRESPONDING VISA FEATURES

The `IPxiSession2` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IPxiSession2`.

Property Name	**VISA Attribute Name**
`SlotWidth`	`VI_ATTR_PXI_SLOT_WIDTH`
`SlotOffset`	`VI_ATTR_PXI_SLOT_OFFSET`

IMPLEMENTATION

RULE 11.2.4

.NET Framework versions of VISA.NET I/O PXI session classes **SHALL** implement `IPxiSession2` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.2.5

.NET Framework versions of VISA.NET I/O session classes that implement the PXI resources **SHALL** implement the interface IPxiSession2.

11.3. ISerialSession

DESCRIPTION

The INSTR session type for serial (RS-232) devices.

DEFINITION

```
public interface ISerialSession : IMessageBasedSession
{
   Int32 BytesAvailable { get; }
   Int32 BaudRate { get; set; }
   LineState ClearToSendState { get; }
   Int16 DataBits { get; set; }
   LineState DataCarrierDetectState { get; }
   LineState DataSetReadyState { get; }
   LineState DataTerminalReadyState { get; set; }
   SerialFlowControlModes FlowControl { get; set; }
   SerialParity Parity { get; set; }
   SerialTerminationMethod ReadTermination { get; set; }
   Byte ReplacementCharacter { get; set; }
   LineState RequestToSendState { get; set; }
   LineState RingIndicatorState { get; }
   SerialStopBitsMode StopBits { get; set; }
   SerialTerminationMethod WriteTermination { get; set; }
   Byte XOffCharacter { get; set; }
   Byte XOnCharacter { get; set; }

   void Flush(IOBuffers buffers, Boolean discard);
   Boolean SetBufferSize(IOBuffers buffers, Int32 size); }
```

CORRESPONDING VISA FEATURES

The `ISerialSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `ISerialSession`.

Property Name	VISA Attribute Name
BytesAvailable	VI_ATTR_ASRL_AVAIL_NUM
BaudRate	VI_ATTR_ASRL_BAUD
ClearToSendState	VI_ATTR_ASRL_CTS_STATE
DataBits	VI_ATTR_ASRL_DATA_BITS
DataCarrierDetectState	VI_ATTR_ASRL_DCD_STATE
DataSetReadyState	VI_ATTR_ASRL_DSR_STATE
DataTerminalReadyState	VI_ATTR_ASRL_DTR_STATE
FlowControl	VI_ATTR_ASRL_FLOW_CNTRL
Parity	VI_ATTR_ASRL_PARITY
ReadTermination	VI_ATTR_ASRL_END_IN
ReplacementCharacter	VI_ATTR_ASRL_REPLACE_CHAR
RequestToSendState	VI_ATTR_ASRL_RTS_STATE
RingIndicatorState	VI_ATTR_ASRL_RI_STATE
StopBits	VI_ATTR_ASRL_STOP_BITS

`WriteTermination`	`VI_ATTR_ASRL_END_OUT`
`XOffCharacter`	`VI_ATTR_ASRL_XOFF_CHAR`
`XOnCharacter`	`VI_ATTR_ASRL_XON_CHAR`

The `ISerialSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `ISerialSession`.

Method Name	**VISA Function Name**
`Flush`	`viFlush`
`SetBufferSize`	`viSetBuf`

IMPLEMENTATION

RULE 11.3.1

VISA.NET I/O ASRL INSTR session classes **SHALL** implement `ISerialSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.3.2

All VISA.NET I/O session classes that implement the ASRL INSTR resource **SHALL** implement the interface `ISerialSession`.

11.4. ITcpipSession

DESCRIPTION

The INSTR session type for LAN devices.

DEFINITION

```
public interface ITcpipSession : IMessageBasedSession
{
   String Address { get; }
   String DeviceName { get; }
   String HostName { get; }
   Int16 Port { get; }
   Boolean IsHiSLIP { get; }
   Boolean HiSLIPOverlapEnabled { get; set; }
   Version HiSLIPProtocolVersion { get; }
   Int32 HiSLIPMaximumMessageKBytes { get; set; }

#if NET6_0_OR_GREATER
   Boolean EncryptionEnabled { get; set; }
   String SaslMechanism { get; }
   String ServerCertificate { get; }
   System.DateTime ServerCertificateExpirationDate { get; }
   Boolean ServerCertificateIsPerpetual { get; }
   String ServerCertificateIssuerName { get; }
   String ServerCertificateSubjectName { get; }
   String TlsCipherSuite { get; }
#endif

   Boolean SetBufferSize(IOBuffers buffers, Int32 size);
   void SendRemoteLocalCommand(RemoteLocalMode mode);
}
```

CORRESPONDING VISA FEATURES

The `ITcpipSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `ITcpipSession`.

Property Name	VISA Attribute Name
Address	VI_ATTR_TCPIP_ADDR
DeviceName	VI_ATTR_TCPIP_DEVICE_NAME
HostName	VI_ATTR_TCPIP_HOSTNAME
Port	VI_ATTR_TCPIP_PORT
IsHiSLIP	VI_ATTR_TCPIP_IS_HISLIP
HiSLIPProtocolVersion	VI_ATTR_TCPIP_HISLIP_VERSION
HiSLIPMaximumMessageKBytes	VI_ATTR_TCPIP_HISLIP_MAX_MESSAGE_KB
HiSLIPOverlapEnabled	VI_ATTR_TCPIP_HISLIP_OVERLAP_EN
EncryptionEnabled	VI_ATTR_TCPIP_HISLIP_ENCRYPTION_ENABLED
SaslMechanism	VI_ATTR_TCPIP_SASL_MECHANISM
ServerCertificate	VI_ATTR_TCPIP_SERVER_CERT
ServerCertificateExpirationDate	VI_ATTR_TCPIP_SERVER_CERT_EXPIRATION_DATE

ServerCertificateIsPerpetual	VI_ATTR_TCPIP_SERVER_CERT_IS_PERPETUAL
ServerCertificateIssuerName	VI_ATTR_TCPIP_SERVER_CERT_ISSUER_NAME
ServerCertificateSubjectName	VI_ATTR_TCPIP_SERVER_CERT_SUBJECT_NAME
TlsCipherSuite	VI_ATTR_TCPIP_TLS_CIPHER_SUITE

The ITcpipSession interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for ITcpipSession.

Method Name	VISA Function Name
SetBufferSize	viSetBuf
SendRemoteLocalCommand	viGpibControlRen

IMPLEMENTATION

OBSERVATION 11.4.1

Security related properties EncryptionEnabled, SaslMechanism, ServerCertificate, ServerCertificateExpirationDate, ServerCertificateIsPerpetual, ServerCertificateIssuerName, ServerCertificateSubjectName, and TlsCipherSuite are omitted from the .NET Framework API for this interface. In the .NET Framework API, they are located in IPxiSession2.

RULE 11.4.2

VISA.NET I/O TCPIP INSTR session classes **SHALL** implement ITcpipSession interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.4.3

All VISA.NET I/O session classes that implement the TCPIP INSTR resource **SHALL** implement the interface ITcpipSession.

OBSERVATION 11.4.2

.NET Framework classes implement ITcpipSession2, which declares the security properties enumerated in OBSERVATION 11.4.1. Because of this, .NET Framework TCPIP session classes must implement the security properties, and so are functionally equivalent to .NET TCPIP session classes that only implement ITcpipSession.

RULE 11.4.4

For implementations of ITcpipSession for VXI-11 devices, IsHiSLIP **SHALL** return false.

OBSERVATION 11.4.3

If IsHiSLIP returns false, accessing the following properties may result in an exception: Port, HiSLIPProtocolVersion, HiSLIPMaximumMessageKBytes, HiSLIPOverlapEnabled.

11.4.2. ITcpipSession2 (.NET Framework Only)

DESCRIPTION

In the .NET Framework API only, the INSTR session type for LAN devices. This derives from and supercedes ITcpipSession for the .NET Framework API only.

DEFINITION

```
#if NETFRAMEWORK
public interface ITcpipSession2 : ITcpipSession
{
   Boolean EncryptionEnabled { get; set; }
```

```
    String SaslMechanism { get; }
    String ServerCertificate { get; }
    System.DateTime ServerCertificateExpirationDate { get; }
    Boolean ServerCertificateIsPerpetual { get; }
    String ServerCertificateIssuerName { get; }
    String ServerCertificateSubjectName { get; }
    String TlsCipherSuite { get; }
}
#endif
```

CORRESPONDING VISA FEATURES

The `ITcpipSession2` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `ITcpipSession2`.

Property Name	**VISA Attribute Name**
`EncryptionEnabled`	`VI_ATTR_TCPIP_HISLIP_ENCRYPTION_ENABLED`
`SaslMechanism`	`VI_ATTR_TCPIP_SASL_MECHANISM`
`ServerCertificate`	`VI_ATTR_TCPIP_SERVER_CERT`
`ServerCertificateExpirationDate`	`VI_ATTR_TCPIP_SERVER_CERT_EXPIRATION_DATE`
`ServerCertificateIsPerpetual`	`VI_ATTR_TCPIP_SERVER_CERT_IS_PERPETUAL`
`ServerCertificateIssuerName`	`VI_ATTR_TCPIP_SERVER_CERT_ISSUER_NAME`
`ServerCertificateSubjectName`	`VI_ATTR_TCPIP_SERVER_CERT_SUBJECT_NAME`
`TlsCipherSuite`	`VI_ATTR_TCPIP_TLS_CIPHER_SUITE`

IMPLEMENTATION

RULE 11.4.5

.NET Framework versions of VISA.NET I/O TCPIP INSTR session classes **SHALL** implement `ITcpipSession2` interface properties and methods as specified in VPP 4.3 for corresponding attributes, except as specified otherwise in this specification.

RULE 11.4.6

.NET Framework versions of VISA.NET I/O session classes that implement the TCPIP INSTR resource **SHALL** implement the interface ITcpipSession2.

OBSERVATION 11.4.4

If `ITcpipSession.IsHiSLIP` returns false, accessing the `EncryptionEnabled` property may result in an exception.

11.5. IUsbSession

DESCRIPTION

The INSTR session type for USBTMC devices.

DEFINITION

```
public interface IUsbSession : IMessageBasedSession
{
   event EventHandler<UsbInterruptEventArgs> Interrupt;

   Boolean Is4882Compliant { get; }
   Int16 MaximumInterruptSize { get; set; }
   Int16 ManufacturerId { get; }
   String ManufacturerName { get; }
   Int16 ModelCode { get; }
   String ModelName { get; }
   Int16 UsbInterfaceNumber { get; }
   Int16 UsbProtocol { get; }
   String UsbSerialNumber { get; }

   Byte[] ControlIn(Int16 requestType,
                    Int16 request,
                    Int16 value,
                    Int16 index,
                    Int16 length);

   void ControlOut(Int16 requestType,
                   Int16 request,
                   Int16 value,
                   Int16 index);
   void ControlOut(Int16 requestType,
                   Int16 request,
                   Int16 value,
                   Int16 index,
                   Byte[] data);

   void SendRemoteLocalCommand(RemoteLocalMode mode);
}
```

CORRESPONDING VISA FEATURES

The `IUsbSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IUsbSession`.

Property Name	**VISA Attribute Name**
`Is4882Compliant`	`VI_ATTR_4882_COMPLIANT`
`MaximumInterruptSize`	`VI_ATTR_USB_MAX_INTR_SIZE`
`ManufacturerId`	`VI_ATTR_MANF_ID`
`ManufacturerName`	`VI_ATTR_MANF_NAME`
`ModelCode`	`VI_ATTR_MODEL_CODE`
`ModelName`	`VI_ATTR_MODEL_NAME`

UsbInterfaceNumber	VI_ATTR_USB_INTFC_NUM
UsbProtocol	VI_ATTR_USB_PROTOCOL
UsbSerialNumber	VI_ATTR_USB_SERIAL_NUM

The IUsbSession interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for IUsbSession.

Method Name	**VISA Function Name**
ControlIn	viUsbControlIn
ControlOut	viUsbControlOut
SendRemoteLocalCommand	viGpibControlREN

The IUsbSession interface has one .NET event that corresponds to an event defined in VISA. The following table shows correspondence for IUsbSession.

Event Name	**VISA Event Name**
Interrupt	VI_EVENT_USB_INTR

IMPLEMENTATION

RULE 11.5.1

VISA.NET I/O USB INSTR session classes **SHALL** implement IUsbSession interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

RULE 11.5.2

All VISA.NET I/O session classes that implement the USB INSTR resource **SHALL** implement the interface IUsbSession.

11.6. IVxiSession

DESCRIPTION

The INSTR session type for VXI devices.

DEFINITION

```
public interface IVxiSession : IMessageBasedSession, IRegisterBasedSession
{
   event EventHandler<VxiInterruptEventArgs> Interrupt;
   event EventHandler<VxiSignalProcessorEventArgs> SignalProcessor;
   event EventHandler<VxiTriggerEventArgs> Trigger;

   Int16 CommanderLogicalAddress { get; }
   VxiAccessPrivilege DestinationAccessPrivilege { get; set; }
   ByteOrder DestinationByteOrder { get; set; }
   VxiDeviceClass DeviceClass { get; }
   Int16 FastDataChannelNumber { get; set; }
   Boolean FastDataChannelUseStreaming { get; set; }
   Boolean FastDataChannelUsePair { get; set; }
   Boolean Is4882Compliant { get; }
   Boolean IsImmediateServant { get; }
   Int16 LogicalAddress { get; }
   Int16 ChassisLogicalAddress { get; }
   Int16 ManufacturerId { get; }
   String ManufacturerName { get; }
   VxiAccessPrivilege MemoryMapAccessPrivilege { get; set; }
   ByteOrder MemoryMapByteOrder { get; set; }
   Int64 MemoryBase { get; }
   Int64 MemorySize { get; }
   AddressSpace MemorySpace { get; }
   ByteOrder SourceByteOrder { get; set; }
   Int16 ModelCode { get; }
   String ModelName { get; }
   Int16 Slot { get; }
   VxiAccessPrivilege SourceAccessPrivilege { get; set; }
   TriggerLine TriggerLine { get; set; }
   TriggerLines TriggerSupport { get; }

   void AssertTrigger(VxiTriggerProtocol protocol);
   Int32 CommandQuery(VxiCommandMode mode, Int32 command);
   Int64 MemoryAllocate(Int64 size);
   void MemoryFree(Int64 offset);}
```

CORRESPONDING VISA FEATURES

The `IVxiSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IVxiSession`.

Property Name	VISA Attribute Name
`CommanderLogicalAddress`	`VI_ATTR_CMDR_LA`
`DestinationAccessPriviledge`	`VI_ATTR_DEST_ACCESS_PRIV`

DestinationByteOrder	VI_ATTR_DEST_BYTE_ORDER
DeviceClass	VI_ATTR_VXI_DEV_CLASS
FastDataChannelNumber	VI_ATTR_FDC_CHNL
FastDataChannelUseStreaming	VI_ATTR_FDC_MODE
FastDataChannelUsePair	VI_ATTR_FDC_USE_PAIR
Is4882Compliant	VI_ATTR_4882_COMPLIANT
ImmediateServant	VI_ATTR_IMMEDIATE_SERV
Logical Address	VI_ATTR_VXI_LA
ChassisLogicalAddress	VI_ATTR_MAINFRAME_LA
ManufacturerID	VI_ATTR_MANF_ID
ManufacturerName	VI_ATTR_MANF_NAME
MemoryMapAccessPrivilege	VI_ATTR_WIN_ACCESS_PRIV
MemoryMapByteOrder	VI_ATTR_WIN_BYTE_ORDER
MemoryBase	VI_ATTR_MEM_BASE
MemorySize	VI_ATTR_MEM_SIZE
MemorySpace	VI_ATTR_MEM_SPACE
SourceByteOrder	VI_ATTR_SRC_BYTE_ORDER
ModelCode	VI_ATTR_MODEL_CODE
ModelName	VI_ATTR_MODEL_NAME
Slot	VI_ATTR_SLOT
SourceAccessPrivilege	VI_ATTR_SRC_ACCESS_PRIV
TriggerLine	VI_ATTR_TRIG_ID
TriggerSupport	VI_ATTR_VXI_TRIG_SUPPORT

The `IVxiSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IVxiSession`.

Method Name	**VISA Function Name**
AssertTrigger	viAssertTrigger
CommandQuery	viVxiCommandQuery
MemoryAllocate	viMemAllocEx
MemoryFree	viMemFreeEx

The `IVxiSession` interface has several .NET events that correspond to events defined in VISA. The following table shows correspondence for `IVxiSession`.

Event Name	**VISA Event Name**
Interrupt	VI_EVENT_VXI_VME_INTR
SignalProcessor	VI_EVENT_VXI_SIGP
Trigger	VI_EVENT_TRIG

IMPLEMENTATION

RULE 11.6.1

VISA.NET I/O VXI and GPIB-VXI INSTR session classes **SHALL** implement `IVxiSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

Section 12: MEMACC Resources

The MEMACC session type lets a controller perform memory access operations. It does this by providing the controller with services to access arbitrary registers or memory addresses on memory-mapped buses.

Two MEMACC session types are defined for VISA.NET I/O. The first is for VXI MEMACC resources and GPIB-VXI resources, and the second is for PXI. For VXI, MEMACC sessions access the individual VXI memory spaces on the VXI backplane (A16, A24, A32, A64). For PXI, MEMACC sessions access the physical memory on the PCI bus.

12.1. IPxiMemorySession

DESCRIPTION

The MEMACC session type for PXI devices.

DEFINITION

```
public interface IPxiMemorySession : IRegisterBasedSession
{
   Int64 MemoryAllocate(Int64 size);
   Int64 MemoryAllocate(Int64 size, Boolean require32BitRegion);
   void MemoryFree(Int64 offset);
}
```

CORRESPONDING VISA FEATURES

The `IPxiMemorySession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IPxiMemorySession`.

Method Name	**VISA Function Name**
`MemoryAllocate`	`viMemAlloc` (result must fit in 32 bits.) `viMemAllocEx`
`MemoryFree`	`viMemFreeEx`
`N/A`	`viMoveEx`[1] `viMoveAsyncEx`

IMPLEMENTATION

RULE 12.1.1

VISA.NET I/O PXI MEMACC session classes **SHALL** implement `IPxiMemorySession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

[1] Refer to the footnote in Section 12.2.

12.2. IVxiMemorySession Interface

DESCRIPTION

The MEMACC session type for VXI devices.

DEFINITION

```
public interface IVxiMemorySession : IRegisterBasedSession
{
   Int16 LogicalAddress { get; }
   void Move(AddressSpace sourceSpace,
             Int64 sourceOffset,
             DataWidth sourceWidth,
             AddressSpace destinationSpace,
             Int64 destinationOffset,
             DataWidth destinationWidth,
             Int64 sourceCount);
}
```

CORRESPONDING VISA FEATURES

The `IVxiMemorySession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IVxiMemorySession`.

Property Name	VISA Attribute Name
`LogicalAddress`	`VI_ATTR_VXI_LA`

The `IVxiMemorySession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IVxiMemorySession`.

Method Name	VISA Function Name
`Move`	`viMoveEx`[1]
`N/A`	`viMoveAsyncEx`[2]

IMPLEMENTATION

RULE 12.2.1

VISA.NET I/O VXI MEMACC session classes **SHALL** implement `IVxiMemorySession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

[1] The intent of this method in this interface is to move data from one device to another. The same is not true of PXI, where such moves are not defined. Moves from a device to local space can be accomplished with MoveIn or MoveOut. The decision not to include Move in IPxiSession or IPxiMemorySession was a deliberate one.

[2] viMoveAsync was deliberately omitted from this interface because it is not a common use case. The equivalent behavior can be controlled with more precision by a multi-threaded client.

Section 13: INTFC Resources

The only INTFC session type defined for VISA.NET I/O resources is the GPIB INTFC resource. The INTFC session type lets a GPIB controller interact with any devices connected to the board associated with this session type. 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.

13.1. IGpibInterfaceSession Interface

DESCRIPTION

The INTFC session type for GPIB buses.

DEFINITION

```
public interface IGpibInterfaceSession : IVisaSession
{
   event EventHandler<VisaEventArgs> Cleared;
   event EventHandler<GpibControllerInChargeEventArgs> ControllerInCharge;
   event EventHandler<VisaEventArgs> Listen;
   event EventHandler<VisaEventArgs> ServiceRequest;
   event EventHandler<VisaEventArgs> Talk;
   event EventHandler<VisaEventArgs> Trigger;

   GpibAddressedState AddressState { get; }
   Boolean AllowDma { get; set; }
   LineState AtnState { get; }
   Int16 HS488CableLength { get; set; }
   Byte DeviceStatusByte { get; set; }
   IOProtocol IOProtocol { get; set; }
   Boolean IsControllerInCharge { get; }
   Boolean IsSystemController { get; set; }
   LineState NdacState { get; }
   Int16 PrimaryAddress { get; set; }
   LineState RenState { get; }
   Int16 SecondaryAddress { get; set; }
   public bool SendEndEnabled { get; set; }
   LineState SrqState { get; }
   public byte TerminationCharacter { get; set; }
   public bool TerminationCharacterEnabled { get; set; }

   public void AssertTrigger();
   void PassControl(Int16 primaryAddress);
   void PassControl(Int16 primaryAddress, Int16 secondaryAddress);
   void ControlAtn(AtnMode command);
   Int32 SendCommand(Byte[] data);
   void SendRemoteLocalCommand(GpibInterfaceRemoteLocalMode mode);
   void SendInterfaceClear();


   IMessageBasedRawIO RawIO { get; }
```

```
}
```

CORRESPONDING VISA FEATURES

The `IGpibInterfaceSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IGpibInterfaceSession`.

Property Name	**VISA Attribute Name**
`AddressingState`	`VI_ATTR_GPIB_ADDR_STATE`
`AllowDma`	`VI_ATTR_DMA_ALLOW_EN`
`AtnState`	`VI_ATTR_GPIB_ATN_STATE`
`DeviceStatusByte`	`VI_ATTR_DEV_STATUS_BYTE`
`IOProtocol`	`VI_ATTR_IO_PROT`
`IsControllerInCharge`	`VI_ATTR_GPIB_CIC_STATE`
`IsSystemController`	`VI_ATTR_GPIB_SYS_CNTRL_STATE`
`HS488CableLength`	`VI_ATTR_GPIB_HS488_CBL_LEN`
`NdacState`	`VI_ATTR_GPIB_NDAC_STATE`
`PrimaryAddress`	`VI_ATTR_GPIB_PRIMARY_ADDR`
`RenState`	`VI_ATTR_GPIB_REN_STATE`
`SecondaryAddress`	`VI_ATTR_GPIB_SECONDARY_ADDR`
`SrqState`	`VI_ATTR_GPIB_SRQ_STATE`

The `IGpibInterfaceSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IGpibInterfaceSession`.

Method Name	**VISA Function Name**
`SendCommand`	`viGpibCommand`
`ControlAtn`	`viGpibControlATN`
`SendRemoteLocalCommand`	`viGpibControlREN`
`PassControl`	`viGpibPassControl`
`SendInterfaceClear`	`viGpibSendIFC`

The `IGpibInterfaceSession` interface has several .NET events that correspond to events defined in VISA. The following table shows correspondence for `IGpibInterfaceSession`.

Event Name	**VISA Function Name**
`Cleared`	`VI_EVENT_CLEAR`
`ControllerInCharge`	`VI_EVENT_GPIB_CIC`
`Listen`	`VI_EVENT_GPIB_LISTEN`
`ServiceRequest`	`VI_EVENT_SERVICE_REQ`
`Talk`	`VI_EVENT_GPIB_TALK`
`Trigger`	`VI_EVENT_TRIG`

IMPLEMENTATION

RULE 13.1.1

VISA.NET I/O GPIB INTFC session classes **SHALL** implement `IGpibInterfaceSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

Section 14: SOCKET Resources

The SOCKET session type 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.

14.1. ITcpipSocketSession

DESCRIPTION

The SOCKET session type for TCPIP devices.

DEFINITION

```
public interface ITcpipSocketSession : IMessageBasedSession
{
   String Address { get; }
   String HostName { get; }
   Boolean KeepAlive { get; set; }
   Boolean NoDelay { get; set; }
   Int16 Port { get; }

#if NET6_0_OR_GREATER
   String ServerCertificate { get; }
   System.DateTime ServerCertificateExpirationDate { get; }
   Boolean ServerCertificateIsPerpetual { get; }
   String ServerCertificateIssuerName { get; }
   String ServerCertificateSubjectName { get; }
   String TlsCipherSuite { get; }
#endif

   void Flush(IOBuffers buffers, Boolean discard);
   Boolean SetBufferSize(IOBuffers buffers, Int32 size);
}
```

CORRESPONDING VISA FEATURES

The `ITcpipSocketSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `ITcpipSocketSession`.

Property Name	VISA Attribute Name
Address	VI_ATTR_TCPIP_ADDR
HostName	VI_ATTR_TCPIP_HOSTNAME
KeepAlive	VI_ATTR_TCPIP_KEEPALIVE
NoDelay	VI_ATTR_TCPIP_NODELAY
Port	VI_ATTR_TCPIP_PORT
ServerCertificate	VI_ATTR_TCPIP_SERVER_CERT
ServerCertificateExpirationDate	VI_ATTR_TCPIP_SERVER_CERT_EXPIRATION_DATE
ServerCertificateIsPerpetual	VI_ATTR_TCPIP_SERVER_CERT_IS_PERPETUAL
ServerCertificateIssuerName	VI_ATTR_TCPIP_SERVER_CERT_ISSUER_NAME
ServerCertificateSubjectName	VI_ATTR_TCPIP_SERVER_CERT_SUBJECT_NAME

TlsCipherSuite	VI_ATTR_TCPIP_TLS_CIPHER_SUITE

The ITcpipSocketSession interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for ITcpipSocketSession.

Method Name	VISA Function Name
Flush	viFlush()
SetBufferSize	viSetBuf()

IMPLEMENTATION

OBSERVATION 14.1.1

Security related properties ServerCertificate, ServerCertificateExpirationDate, ServerCertificateIsPerpetual, ServerCertificateIssuerName, ServerCertificateSubjectName, and TlsCipherSuite are omitted from the .NET Framework API for this interface. In the .NET Framework API, they are located in ITcpipSocketSession2.

RULE 14.1.2

VISA.NET I/O TCPIP SOCKET session classes **SHALL** implement ITcpipSocketSession interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

14.1.2. ITcpipSocketSession2 (.NET Framework Only)

DESCRIPTION

In the .NET Framework API only, the SOCKET session type for TCPIP devices. This derives from and supercedes ITcpipSocketSession for the .NET Framework API only,.

DEFINITION

```
#if NETFRAMEWORK
public interface ITcpipSocketSession2 : ITcpipSocketSession
{
   String ServerCertificate { get; }
   System.DateTime ServerCertificateExpirationDate { get; }
   Boolean ServerCertificateIsPerpetual { get; }
   String ServerCertificateIssuerName { get; }
   String ServerCertificateSubjectName { get; }
   String TlsCipherSuite { get; }
}
#endif
```

CORRESPONDING VISA FEATURES

The ITcpipSocketSession2 interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for ITcpipSocketSession2.

Property Name	VISA Attribute Name
ServerCertificate	VI_ATTR_TCPIP_SERVER_CERT
ServerCertificateExpirationDate	VI_ATTR_TCPIP_SERVER_CERT_EXPIRATION_DATE
ServerCertificateIsPerpetual	VI_ATTR_TCPIP_SERVER_CERT_IS_PERPETUAL
ServerCertificateIssuerName	VI_ATTR_TCPIP_SERVER_CERT_ISSUER_NAME
ServerCertificateSubjectName	VI_ATTR_TCPIP_SERVER_CERT_SUBJECT_NAME

`TlsCipherSuite`	`VI_ATTR_TCPIP_TLS_CIPHER_SUITE`

IMPLEMENTATION

RULE 14.1.3

.NET Framework versions of VISA.NET I/O TCPIP SOCKET session classes **SHALL** implement `ITcpipSocketSession2` interface properties and methods as specified in VPP 4.3 for corresponding attributes, except as specified otherwise in this specification.

RULE 14.1.4

.NET Framework versions of VISA.NET I/O session classes that implement the TCPIP INSTR resource **SHALL** implement the interface ITcpipSocketSession2.

Section 15: BACKPLANE Resources

The BACKPLANE session type 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.

There is generally one BACKPLANE resource per configured chassis.

Backplane session types differ from other session types in that they provide no communication (messaging or register) operations.

15.1. IPxiBackplaneSession

DESCRIPTION

The BACKPLANE session type for PXI backplanes.

DEFINITION

```
public interface IPxiBackplaneSession : IVisaSession
{
   Int16 ChassisNumber { get; }
   String ManufacturerName { get; }
   String ModelName { get; }

   void ReserveTrigger(Int16 bus, TriggerLine line);

   void ReserveTriggers(Int16[] buses, TriggerLine[] lines);

   void UnreserveTrigger(Int16 bus, TriggerLine line);
   void MapTrigger(Int16 sourceBus, TriggerLine sourceLine,
                   Int16 destinationBus, TriggerLine destinationLine);
   void MapTrigger(Int16 sourceBus, TriggerLine sourceLine,
                   Int16 destinationBus, TriggerLine destinationLine,
                   out Boolean alreadyMapped);
   void UnmapTrigger(Int16 sourceBus, TriggerLine sourceLine);
   void UnmapTrigger(Int16 sourceBus, TriggerLine sourceLine,
                     Int16 destinationBus, TriggerLine destinationLine);
}
```

CORRESPONDING VISA FEATURES

The `IPxiBackplaneSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IPxiBackplaneSession`.

Property Name	VISA Attribute Name
`ChassisNumber`	`VI_ATTR_PXI_CHASSIS`
`ManufacturerName`	`VI_ATTR_MANF_NAME`
`ModelName`	`VI_ATTR_MODEL_NAME`

The `IPxiBackplaneSession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IPxiBackplaneSession`.

Method Name	VISA Function Name
`ReserveTrigger`	`viPxiReserveTriggers`
`ReserveTriggers`	`viPxiReserveTriggers`
`UnreserveTrigger`	`viAssertTrigger` w/ `VI_TRIG_PROT_UNRESERVE`
`MapTrigger`	`viMapTrigger`
`UnmapTrigger`	`viUnmapTrigger`

IMPLEMENTATION

RULE 15.1.1

VISA.NET I/O PXI BACKPLANE session classes **SHALL** implement `IPxiBackplaneSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

15.2. IVxiBackplaneSession

DESCRIPTION

The BACKPLANE session type for VXI backplanes.

DEFINITION

```
public interface IVxiBackplaneSession : IVisaSession
{
   event EventHandler<VxiTriggerEventArgs> Trigger;
   event EventHandler<VisaEventArgs> SystemFailure;
   event EventHandler<VisaEventArgs> SystemReset;

   Int16 ChassisLogicalAddress { get; }
   TriggerLines TriggerStatus { get; }
   TriggerLines TriggerSupport { get; }
   Int16 InterruptStatus { get; }
   LineState SystemFailureStatus { get; }

   void AssertInterrupt(Int16 irqLevel, Int32 statusId);
   void AssertTrigger(TriggerLine line, VxiTriggerProtocol protocol);
   void AssertUtilitySignal(VxiUtilitySignal signal);
   void MapTrigger(TriggerLine sourceLine, TriggerLine destinationLine);
   void MapTrigger(TriggerLine sourceLine, TriggerLine destinationLine,
                   out Boolean alreadyMapped);
   void UnmapTrigger(TriggerLine sourceLine);
   void UnmapTrigger(TriggerLine sourceLine, TriggerLine destinationLine);
}
```

CORRESPONDING VISA FEATURES

The `IVxiBackplaneSession` interface has several .NET properties that correspond to attributes defined in VISA. The following table shows property-attribute equivalence for `IVxiBackplaneSession`.

Property Name	**VISA Attribute Name**
`ChassisLogicalAddress`	`VI_ATTR_MAINFRAME_LA`
`TriggerLine`[1]	`VI_ATTR_TRIG_ID`
`TriggerStatus`	`VI_ATTR_VXI_TRIG_STATUS`
`TriggerSupport`	`VI_ATTR_VXI_TRIG_SUPPORT`
`InterruptStatus`	`VI_ATTR_VXI_VME_INTR_STATUS`
`SystemFailureStatus`	`VI_ATTR_VXI_VME_SYSFAIL_STATE`

The `IVxiBackplaneession` interface has several .NET methods that correspond to functions defined in VISA. The following table shows method-function correspondence for `IVxiBackplaneSession`.

Method Name	**VISA Function Name**
`AssertInterruptSignal`	`viAssertIntrSignal`
`AssertTrigger`	Set `VI_ATTR_TRIG_ID` and then execute `viAssertTrigger`
`AssertUtilSignal`	`viAssertUtilSignal`

[1] Refer to 0 on why this is not listed in `IVxiBackplaneSession`.

`MapTrigger`	`viMapTrigger`
`UnmapTrigger`	`viUnmapTrigger`

The `IVxiBackplaneSession` interface has several .NET events that correspond to events defined in VISA. The following table shows correspondence for `IVxiBackplaneSession`.

Event Name	**VISA Function Name**
`SystemFailure`	`VI_EVENT_VXI_VME_SYSFAIL`
`SystemReset`	`VI_EVENT_VXI_VME_SYSRESET`
`Trigger`	`VI_EVENT_TRIG`

IMPLEMENTATION

RULE 15.2.1

VISA.NET I/O VXI BACKPLANE session classes SHALL implement `IVxiBackplaneSession` interface properties and methods as specified in VPP 4.3 for corresponding attributes and functions, except as specified otherwise in this specification.

OBSERVATION 15.2.1

`TriggerLine` was accidentally omitted from IVxiBackplaneSession. Vendor implementations are encouraged to provide the `TriggerLine` property in a class that implements `IVxiBackplaneSession`. `TriggerLine` may be added to `IVxiBackplaneSession` in a future version of the specification.

Section 16: VISA.NET I/O Conflict Resolution

In cases where more than one vendor-specific VISA.NET library can connect to an interface, the conflict resolution manager provides information regarding available vendor-specific VISA.NET libraries and user preferences. It also provides the same services for C (64-bit) and COM implementations.

There is one implementation of the conflict resolution manager for VISA C, COM, and .NET. This implementation is provided by the IVI Foundation and installed as part of the VISA Shared Components. The behavior and both the C and .NET APIs are described in *VPP-4.3.5: VISA Shared Components.*

VISA.NET conflict resolution information is used by the VISA.NET Global Resource Manager (GRM), which is described in more detail in section 17.2,*IResourceManager Interface*. The Conflict Manager API may be used by vendor-specific utilities or user programs to maintain conflict resolution information.

Note that there are installation requirements for VISA.NET implementations, which enable the implementations to be managed by the conflict manager. These requirements are described in Section 18.2 *Vendor-Specific VISA.NET Installer Requirements*.

Section 17: Resource Manager Classes

Each VISA.NET session class must include a constructor that creates a session and initializes a VISA.NET I/O Resource. However, the recommended way to create the session is to use a VISA.NET *resource manager*. (Note that this provides a consistent way to instantiate session classes, since the signature for session class constructors is not specified.) There are two types of resource manager, vendor specific resource managers and the VISA.NET Shared Components Global Resource Manager, or GRM.

Vendor specific resource managers are provided as part of a particular vendor's implementation of VISA.NET. A vendor specific resource manager knows what session types can be instantiated by the implementation, what resource descriptors will be recognized, and what sessions can actually be instantiated. Its most important capability, however, is that it can instantiate and return a session that allows communication with a resource.

The vendor specific resource managers implement the `IResourceManager` interface, so that the API for all vendor specific resource managers is standard. `IResourceManager` references to various resource managers may also be interchanged.

The VISA.NET Shared Components GRM is a static class that is one of the VISA.NET shared components. It ***does not*** know what session types can be instantiated by the implementation, what resource descriptors will be recognized, and thus what sessions can actually be instantiated. However, it ***does*** know how to query vendor specific resource managers to discover this information. It can be used when multiple implementations of VISA.NET are installed to consolidate information from all of them and select one to instantiate a session. Note that while the methods and properties defined by the GRM correspond in name to those defined by `IResourceManager`, there are some differences in parameters, and all of the GRM methods and properties are static.

This section also includes the definition of the `ParseResult` class, which consolidates all of the information returned by the `Parse` methods into one object.

17.1. The Vendor-Specific Resource Manager Component

Vendor specific resource managers are provided as part of a particular vendor's implementation of VISA.NET. Each vendor specific resource manager derives from IResourceManager and includes a public constructor with no parameters for use by the GRM.

RULE 17.1.1

A vendor-specific resource manager component **SHALL** be implemented as a non-static class. The class **SHALL** derive from IResourceManager.

RULE 17.1.2

Vendor Specific RMs **SHALL** have a public constructor with no parameters. They may have other constructors.

RULE 17.1.3

A vendor-specific resource manager component SHALL be able to create instances of one or more session classes provided by that vendor.

RULE 17.1.4

There SHALL be exactly one Vendor specific RM per registered assembly qualified name.

PERMISSION 17.1.1

There may be more than one vendor-specific resource manager for a particular session component.

RULE 17.1.5

The ImplementationVersion property of the vendor-specific manager SHALL relate to the VISA attribute VI_ATTR_RSRC_IMPL_VERSION as follows.

- Major value is treated the same as .NET MajorVersion.
- Minor value is treated the same as .NET MinorVersion.
- .NET Build and Revision - build.revision is monotonically increasing.

OBSERVATION 17.1.1

The ImplementationVersion, SpecificationVersion, ManufacturerID, and ManufacturerName properties reflect the VISA.NET implementation. If there is an underlying VISA C I/O implementation, these properties need not reflect the corresponding values of the underlying VISA C I/O.

RULE 17.1.6

The SpecificationVersion property SHALL be identical to the version of the specification with which the shared components used conform. Build and revision **SHALL** both be zero.

RULE 17.1.7

The set of resources returned by Find SHALL be identical to the set returned in VISA by a call to viFindRsrc followed by viFindNext until all discovered resources are found.

RULE 17.1.8

The vendor-specific Parse method SHALL have the same behavior as the viParseRsrcEx method described in VPP-4.3 with the following exceptions.

- The vendor-specific Parse **SHALL** understand resource strings for only the interface types, session types, and interface numbers for which it provides an implementation.
- The Parse method **SHALL** NOT perform operations that would affect other operations in progress on the resource.

RULE 17.1.9

IF a vendor-specific resource manager can create any particular resource on a given hardware interface, THEN it SHALL be capable of creating all available resources on that interface. The vendor-specific resource manager's Find() method, with an pattern argument equal to "*", will return all of the available resources for the vendor's VISA.NET implementation.

RULE 17.1.10

The vendor-specific resource manager SHALL be registered as described in Section 18.2.3, *VISA.NET Registry Entries*.

17.2. IResourceManager Interface

DESCRIPTION

The `IResourceManager` interface provides methods that instantiate a VISA.NET session for the specified resource, parse resource names and return the individual pieces of information that they conveys, and find the resources (by resource name) configured by VISA.NET that match the specified pattern.

DEFINITION

```
public interface IResourceManager : IDisposable
{
   IEnumerable<String> Find(String pattern);
   ParseResult Parse(String resourceName);
   IVisaSession Open(String resourceName);
   IVisaSession Open(String resourceName,
                     AccessModes accessMode,
                     Int32 timeoutMilliseconds);
   IVisaSession Open(String resourceName,
                     AccessModes accessModes,
                     Int32 timeoutMilliseconds,
                     out ResourceOpenStatus openStatus);
   String ManufacturerName { get; }
   Int16 ManufacturerId { get; }
   Version ImplementationVersion { get; }
   Version SpecificationVersion { get; }
}
```

CORRESPONDING VISA FEATURES

The `IResourceManager` interface methods all map to VISA functions. The following table shows VISA correspondence for `IResourceManager` methods.

Method Name	**VISA Function Name**
`Find()`	`viFindRsrc(), viFindNext()`
`Parse()`	`viParseRsrcEx()`
`Open()`	`viOpen()`

The `IResourceManager` interface properties all map to VISA attributes. The following table shows VISA correspondence for `IResourceManager` properties.

Property Name	**VISA Attribute Name**
`ManufacturerName`	`VI_ATTR_RSRC_MANF_NAME`
`ManufacturerID`	`VI_ATTR_RSRC_MANF_ID`
`ImplementationVersion`	`VI_ATTR_RSRC_IMPL_VERSION`
`SpecificationVersion`	`VI_ATTR_RSRC_SPEC_VERSION`

OBSERVATION 17.2.1

In the VISA C API, `viOpen()` returns a positive value to indicate a warning or to provide additional information about a successful call. The `openStatus` argument to `Open()` is used to indicate the equivalent information. Note that it is an `out` argument.

17.3. The Global Resource Manager (GRM) Component

The Global Resource Manager's (GRM) main responsibilities are locating, instantiating, and using the vendor managers and resolving any overlapping functionality between vendor-specific managers. It is distributed with the VISA.NET Shared Components.

RULE 17.3.1

The ManufacturerName property **SHALL** return "IVI Foundation" and the ManufacturerID property SHALL return 0x3FFF.

RULE 17.3.2

The ImplementationVersion property of the vendor-specific manager SHALL relate to the VISA attribute VI_ATTR_RSRC_IMPL_VERSION as follows.

- Major value is treated the same as .NET MajorVersion.
- Minor value is treated the same as .NET MinorVersion.
- .NET Build and Revision - build.revision is monotonically increasing.

RULE 17.3.3

The SpecificationVersion property **SHALL** be identical to the version of the specification with which the GRM conforms. Build and revision **SHALL** both be zero.

RULE 17.3.4

The Find method **SHALL** call the Find method on all the vendor-specific resource managers. Any resource strings that are equivalent according to the rules defined in *VPP 4.3: The VISA Library*, section 4.3.1, *Address String*, for resource strings **SHALL** be discarded, and a new array of strings with the combined results **SHALL** be returned to the user.

OBSERVATION 17.3.1

The GRM ignores a given RM if the RM implementation is not compatible for any reason with the current runtime & process.

RULE 17.3.5

When an implementation of IResourceManager is released, it **SHALL NOT** cause the sessions that it opened to be closed.

OBSERVATION 17.3.2

The previous rule is inconsistent with the way resource sessions are handled in VISA C. It is a better match to .NET paradigms.

OBSERVATION 17.3.3

GlobalResourceManager methods will not hold references to the vendor specific resource manager sessions that they use to accomplish their tasks after the method exits. This is the reason for RULE 17.3.5.

17.3.2. Vendor VISA.NET Loading

The IVI VISA.NET assembly contains the GRM (Global Resource Manager) that is responsible for loading individual vendor VISA assemblies that provide the actual IO semantics. This dynamic loading requires that the GRM be able to locate the vendor assemblies, and determine that they implement a version of the VISA interfaces that is compatible with the IVI VISA.NET assembly (which was loaded by the client).

This section describes the requirements of vendor VISA libraries, and the required behavior of the IVI VISA.NET GRM to ensure that correct versions are loaded.

17.3.2.1. IVI VISA.NET Version

RULE 17.3.6

The vendor VISAs loaded by the IVI VISA.NET may be built against a different version of the IVI VISA.NET assembly than the version that is attempting to dynamically load them. The version of the IVI VISA upon which the vendor VISA is dependent is used by the GRM to determine if the vendor VISA is compatible as follows:

MajorVersion	The IVI VISA assembly will only load vendor VISA that is dependent on (was compiled against) an IVI VISA assembly of the same MajorVersion. The Major Version is incremented when API changes are incompatible with the previous version. Refer to section 19.3, *Maintaining Backwards Compatibility.*
MinorVersion	If and only if the major version is the same, the IVI VISA assembly will only load a vendor VISA that is dependent on (was compiled against) on IVI VISA assembly with the same or an earlier IVI VISA MinorVersion. The Minor Version is incremented when API changes are backwards compatible with other releases of the same Major verion.

The third field of the IVI VISA.NET version corresponds to the Build and increases with each release. The fourth field is always zero.

OBSERVATION 17.3.4

Following these version rules for the IVI VISA assembly may lead to decoupling the assembly version from the specification version. Historically the specification major version has been more of a external indication of the amount of changes, not reflecting backwards compatibility.

17.3.2.2. Vendor VISA.NET Version

PERMISSION 17.3.1

Vendors are permitted to assign version numbers to their assemblies however they like.

17.3.2.3. IVI Global Resource Manager Algorithm to locate Vendor VISA libraries

Refer to Section 18.1, *Vendor VISA .NET (6+) Installation* for details related to installation of vendor implementations.

The GRM will prefer vendor VISAs that are already loaded and bypass the loading algorithm for assemblies that were loaded by the runtime, or previously loaded by the GRM.

To load a vendor VISA, the GRM will locate and load the vendor VISA as follows:

1. The GRM determines the path to the VISA installation directory.

   If the environment variable <IviVisaVendorAssembliesPath> is not null, it sets the VISA installation directory. If <IviVisaVendorAssembliesPath> is null, then the VISA installation directory is<*VXIPNPPATH64*>*\Microsoft.NET\VendorAssemblies*>. If that directory does not exist, or the registry key/value does not exist, then the path is set to be VisaVendorAssemblies subdirectory of the application bin directory.

   (<VXIPNPPATH64> is the target directory for VISA components. Refer to VPP-4.3.5, section 4.2 for the default value and relevant constraints)

OBSERVATION 17.3.5

Customers can use this environement variable to load vendor assmblies from a location of their choosing on a per-process basis. For instance, an application may deploy with application specific versions of the assemblies.

2. The GRM locates and loads compatible vendor assemblies

The directory path for the VendorAssemblies directory has the following appended to the path to the VISA installation directory from the previous step:

```
…\<vendors designation>\<Dependent VISA version>\<visa libraries>
```

These directories are defined as follows:

Vendor designation This is the 2-character vendor abbreviation assigned in VPP-9. The GRM determines this abbreviation using the *ResourceManufacturerID* property as described in section 8.3, *IVisaSession Interface.*

If the *ResourceManufacturerID* is 0xfff or less, it designates a VXI resource manufacturer ID which the GRM translates to a VPP-9 abbreviation.

If the *ResourceManufacturerID* is greater than 0x6100 it represents the 2-character VPP-9 manufacturer as packed UNICODE 8 (lexical order). Note that only alphabetic characters are valid VPP-9 IDs. These characters shall be lower case (from 0x61 through 0x7a inclusive).

Dependent VISA Version This field indicates the Major.Minor version of the IVI VISA library upon which this vendor VISA depends. Notice that this is not the version of the vendor VISA.

This declaration of the IVI VISA dependency is used by the GRM to determine if a vendor VISA is suitable for use with the running version of the GRM.*VISA Libraries.*

The files in the <Dependent VISA Version> folder are loaded by the GRM to load a vendor VISA. A *deps.json* file must be included to described the dependencies of the vendor VISA that the GRM must load.

Note: that the vendor VISAs are named <*vendor*>.*visa.dll.* The <*vendor*> in this name is not the VPP-9 ID. The GRM will load any vendor *visa.dll* found in this directory.

17.4. GlobalResourceManager Class

DESCRIPTION

The `GlobalResourceManager` class provides methods that instantiate a VISA.NET session for the specified resource, parse resource names and return the individual pieces of information that they convey, and find the resources (by resource name) configured by the vendor specific VISA.NET implementations that match the specified pattern.

DEFINITION

```
public static class GlobalResourceManager
{
   public static IEnumerable<String> Find() {…}
   public static IEnumerable<String> Find(String pattern) {…}
   public static ParseResult Parse(String resourceName) {…}
   public static Boolean TryParse(String resourceName,
                                  out ParseResult result) {…}
   public static IVisaSession Open(String resourceName) {…}
   public static IVisaSession Open(String resourceName,
                                   AccessModes accessMode,
                                   Int32 timeoutMilliseconds) {…}
   public static IVisaSession Open(String resourceName,
                                   AccessModes accessModes,
                                   Int32 timeoutMilliseconds,
                                   out ResourceOpenStatus openStatus) {…}

   public static String ManufacturerName { get; }
   public static Int16 ManufacturerId { get; }
   public static Version ImplementationVersion { get; }
   public static Version SpecificationVersion { get; }
}
```

CORRESPONDING VISA FEATURES

The `IResourceManager` interface methods all map to VISA functions. The following table shows VISA correspondence for `IResourceManager` methods.

Method Name	VISA Method Name
`Find()`	`viFindRsrc(), viFindNext()`
`Open()`	`viOpen()`
`Parse()`	`viParseRsrcEx()`
`TryParse()`	`viParseRsrcEx()`

The `IResourceManager` interface properties all map to VISA attributes. The following table shows VISA correspondence for `IResourceManager` properties.

Property Name	VISA Attribute Name
`ManufacturerName`	`VI_ATTR_RSRC_MANF_NAME`
`ManufacturerID`	`VI_ATTR_RSRC_MANF_ID`
`ImplementationVersion`	`VI_ATTR_RSRC_IMPL_VERSION`
`SpecificationVersion`	`VI_ATTR_RSRC_SPEC_VERSION`

OBSERVATION 17.4.1

In the VISA C API, `viOpen()` returns a positive value to indicate a warning or to provide additional information about a successful call. The `openStatus` argument to `Open()` is used to indicate the equivalent information. Note that it is an `out` argument.

17.5. ParseResult Class

DESCRIPTION

The `ParseResult` class provides the parsing information returned by the `Parse` methods in the `IResourceManager` interface and the `GlobalResourceManager` class.

DEFINITION

```
public class ParseResult
{
   public String OriginalResourceName { get; private set; }
   public HardwareInterfaceType InterfaceType { get; private set; }
   public Int32 InterfaceNumber { get; private set; }
   public String ResourceClass { get; private set; }
   public String ExpandedUnaliasedName { get; private set; }
   public String AliasIfExists { get; private set; }
   public ParseResult(String originalResourceName,
                      HardwareInterfaceType interfaceType,
                      Int16 interfaceNumber,
                      String resourceClass,
                      String expandedUnaliasedName,
                      String aliasIfExists) {…}
   public static Boolean operator ==(ParseResult parse1, ParseResult parse2)
   public static Boolean operator !=(ParseResult parse1, ParseResult parse2)
}
```

CORRESPONDING VISA FEATURES

The `ParseResult` class properties all map to parameters to the VISA `viParseRsrcEx` method. The following table shows the VISA correspondence for `ParseResult` properties.

Property Name	VISA viParseRsrcEx Parameter Name
`OriginalResourceName`	`rsrcName`
`InterfaceType`	`intfType`
`InterfaceNumber`	`intfNum`
`ResourceClass`	`rsrcClass`
`ExpandedUnaliasedName`	`unaliasedExpandedRsrcName`
`AliasIfExists`	`aliasIfExists`

See VPP-4.3.5 for additional details about the Global Resource Manager implementation.

STANDARD .NET FEATURES

Two standard .NET operators, == and !=, are defined to facilitate comparing parse results.

Section 18: VISA.NET Installation

18.1. Vendor VISA .NET (6+) Installation

This section describes the installation requirements for vendor VISA.NET installation. These requirements are heavily driven by the behavior of the GRM per Section 17.3.2, *Vendor VISA.NET Loading*.

Vendors are permitted to use any mechanism they prefer to install their VISA libraries.

The IVI VISA.NET Assembly is provided in a NuGet package as described in VPP-4.3.5. This section describes the installation of vendor VISA.NET assemblies.

RULE 18.1.1

Vendor VISA.NET assemblies and their dependencies shall be installed by the vendor to:

<*VXIPNPPATH64*>\Microsoft.NET\VendorAssemblies\<*vendor*>\<*dependent VISA version*>\<*assembly*>

Where:

- <*VXIPNPPATH64*> is the target directory for VISA components. Refer to VPP-4.3.5, section 4.2 for the default value and relevant constraints.
- <vendor> is the lower-case 2-character abbreviation for the vendor as defined in VPP-9.
- <dependent VISA version> is the Major.Minor version of the IVI.VISA library on which this version of the vendor VISA depends.
- <assembly> is the name of the assembly.

A *deps.json* files must be included to describe any dependencies that are to be loaded by the GRM.

RULE 18.1.2

Vendor should either install, or leave installed, all versions of their VISA.NET library that they support with the installed VISA.C libraries. This will maximize the applicability of the VISA.NET install to clients that may be using earlier versions of the IVI VISA.NET library.

OBSERVATION 18.1.1

Vendors MAY provide nuget packages for their VISA.NET to enable customers to easily acquire and use vendor-specific APIs. Users need to be cautious mixing static references and dynamic references to libraries.

18.1.2. General Installation Requirements for Vendor Specific Components

RULE 18.1.3

Each VISA.NET I/O implementation **SHALL** consist of one Vendor-Specific Resource Manager (SRM) and one or more Session classes.

PERMISSION 18.1.1

A Vendor may provide more than one version of the vendor VISA.NET I/O implementation.

RULE 18.1.4

A vendor's VISA.NET uninstaller or its SRM uninstaller SHALL NOT silently uninstall the IVI VISA.NET Shared Components.

RULE 18.1.5

If a vendor's VISA installer calls the VISA.NET Standard Components installer, it **SHALL** invoke the VISA.NET Standard Components installer with admin privileges.

18.2. VISA.NET Framework Installation

This section covers the installation of the VISA.NET Shared Components and vendor VISA.NET implementations.

For VISA.NET Shared Components, it includes
- Prerequisites
- Files and directories.

For vendor VISA.NET implementations, it includes
- Registry entries that need to be added to identify the implementation.

18.2.1. VISA.NET Shared Components

VISA.NET Shared Components is an IVI Foundation provided installer that provides the common components needed to provide consistency across VISA.NET implementations from multiple vendors. The VISA.NET Shared Components installer is documented in *VPP-4.3.5: VISA Shared Components*.

Multiple versions of the VISA.NET Shared Components may coexist on a system. If there are more than one version installed at once, publisher policy files will redirect references to earlier versions of the VISA.NET assembly to the latest installed version. This behavior may be overridden using application or machine policy files.

18.2.2. Vendor-Specific VISA.NET Installer Requirements

Vendor-specific VISA.NET installers are created by vendors, but must meet the requirements detailed in this section.

18.2.2.1. Prerequisites

The following software must be installed before a vendor-specific VISA.NET implementation is installed.
- VISA.NET Shared Components.

RULE 18.2.1

Vendor-specific VISA.NET installers **SHALL** either install a suitable version of the VISA.NET Shared Components, or require that a suitable version of the VISA.NET Shared Components is installed before making any VISA.NET related modifications to the install PC.

OBSERVATION 18.2.1

Vendor-specific VISA.NET installers may choose whether to install a suitable version of the VISA.NET Shared Components.

18.2.2.2. VISA.NET Implementation Location

In general, vendors are free to install their vendor-specific implementation of VISA.NET wherever they choose. If they choose to install in the standard VISA directory structure, then there are a few requirements that must be observed. For more information regarding the standard VISA directory structure, refer to Section 4.3, *The Directory Structure* in *VPP-6: Installation and Packaging Specification*. Note that the default value for VXIPNPPATH is either "C:\Program Files\IVI Foundation\VISA" or "C:\Program Files (x86)\IVI Foundation\VISA" depending on the OS bitness and execution context.

PERMISSION 18.2.1

Vendors may install vendor-specific VISA.NET files in the directory <VXIPNPPATH>\Microsoft.NET\Framework32\<FrameworkVersion>\<Vendor Name> <Optional Product ID Text> VISA.NET <Version Text>\, where <Version Text> is derived from the installer version.

This directory is known as the *vendor-specific VISA.NET install directory*. Vendors may also install vendor-specific VISA.NET files in directories other than in the <VXIPNPPATH> directory tree.

RULE 18.2.2

The format of the installer version **SHALL** be <*MajorVersion*>.<*MinorVersion*>.<*Build*> (<Revision> does not apply to installers).

RULE 18.2.3

The format of the <Version Text> **SHALL** be either <*MajorVersion*>.<*MinorVersion*> or <*MajorVersion*>.<*MinorVersion*>.<*Build*>, with the value of each field matching the corresponding field of the installer version.

RULE 18.2.4

A vendor **SHALL NOT** install VISA.NET to any location under <IVI_ROOT_DIR>, with the exception of the vendor-specific VISA.NET install directory.

RULE 18.2.5

In the vendor-specific VISA.NET install directory name, <VendorName> **SHALL** be the name reported by the vendor-specific resource managers' ManufacturerName property.

PERMISSION 18.2.2

The directory name may include additional arbitrary text, <Optional Product ID Text>, to distinguish multiple products from the same vendor that provide VISA.NET implementations. This text is optional, and does not need to match the actual product name.

18.2.3. VISA.NET Registry Entries

Vendor-specific VISA.NET installers must register their vendor-specific Resource Manager so that the Global Resource Manager can locate and instantiate it.

RULE 18.2.6

Vendor-specific VISA.NET installers **SHALL** add the registry key `HKLM\SOFTWARE\IVI\VISA.NET\<GUID>\<version>` where `<GUID>` is a GUID that is unique to the vendor's VISA.NET product and <version> is the installer version. This registry entry is in the 32-bit hive.

RULE 18.2.7

For each product that provides an implementation of VISA.NET, there **SHALL** be exactly one GUID in the registry.

OBSERVATION 18.2.2

One vendor may have more than one product that provides an implementation of VISA.NET. In this case, there would be a unique GUID for each product, but not for each version of each product.

RULE 18.2.8

There **SHALL** be exactly one version key for each installed version of a product that provides an implementation of VISA.NET, with the key name in the format specified in RULE 18.2.2.

RULE 18.2.9

The version key **SHALL** have the following values with data types:

- Comments (REG_SZ)
- FriendlyName (REG_SZ)
- VendorID (REG_DWORD)
- Location (REG_SZ)

RULE 18.2.10

The Location and FriendlyName values **SHALL NOT** be empty.

RULE 18.2.11

The Location value **SHALL** be the assembly qualified name of the vendor-specific resource manager class. This name consists of the fully qualified type name of the class and the assembly qualified path, for example, "TmCo.Visa.ResourceManager, TmCo.Visa, Version=1.0.0.0, Culture=neutral, PublicKeyToken=f372f203818f2407, processorArchitecture=MSIL".

RECOMMENDATION 18.2.1

The recommended format for the FriendlyName is "<ManufacturerName> VISA.NET Resource Manager". In some cases a manufacturer may register more than one resource manager, in which case appropriate friendly names may be selected for each one.

RULE 18.2.12

The VendorID **SHALL** match the value of the ManufacturerID property returned by the vendor-specific resource manager referenced by the Location value.

18.2.4. VISA.NET Resource Manager Registration

RULE 18.2.13

The assembly containing the vendor-specific resource manager SHALL be installed into the Global Assembly Cache (GAC).

18.2.4.2. General Installation Requirements for Vendor Specific Components

RULE 18.2.14

Each VISA.NET I/O implementation **SHALL** consist of one Vendor-Specific Resource Manager (SRM) and one or more Session classes.

PERMISSION 18.2.3

A Vendor may provide more than one VISA.NET I/O implementation.

RULE 18.2.15

A vendor's VISA.NET uninstaller or its SRM uninstaller SHALL NOT silently uninstall the VISA.NET Standard Components.

RULE 18.2.16

On Windows 7, Windows 8, Windows 10, and Windows 11, if a vendor's VISA installer calls the VISA.NET Standard Components installer, it **SHALL** invoke the VISA.NET Standard Components installer with admin privileges.

Section 19: .NET Version Control

IVI provides interoperability of VISA.NET implementations from multiple vendors released at various times and without coordination of release schedules between the vendors. At the same time, IVI must periodically revise the IVI VISA.NET assembly to support new features and correct defects.

In principle, IVI versioning is designed to make it possible to create an application that uses different vendors' VISA.NET implementations. The vendor implementations may be created with different versions of the VISA.NET shared components provided by the IVI Foundation. Applications or libraries that use VISA do not need to adopt a special internal architecture to accommodate the different versions. This is key to the interoperability features of VISA.NET.

19.1. Versioning Concepts

All IVI .NET versioning strategies share a basic objective. If IVI creates a new version of a .NET assembly, the new version should be created in a manner which does not make existing applications inoperable. To do this, the APIs in the new version of the assembly must include all the elements of the older version. After all, if parts of the API are deleted, applications or libraries that reference the assembly might suddenly find that new versions of the referenced assembly don't include features that they use. In such cases, .NET will fail to load the assembly, or if it does, will report an exception of some sort. To address this problem, features are never deleted from .NET APIs,[1] so that the newer APIs are backwards compatible with the older ones.

19.2. Versioning Scenarios

Applications or libraries that use the IVI VISA.NET GRM reference a specific version of the IVI VISA.NET assembly. The GRM will attempt to dynamically load a vendor implementation, which also references a specfic version (often a different specific version) of the IVI VISA.NET assembly. If the GRM is built with a reference to a newer VISA.NET assembly, and the vendor implementation is built with a reference to an older assembly, this is a "down-version" scenario. If the GRM is built with a reference to an older VISA.NET assembly, and the vendor implementation is built with a reference to a newer assembly, this is an "up-version" scenario.

.NET Framework and .NET (6+) have different strategies for handling down- and up-versioning, assuming that the APIs are backwards compatible.

- .NET Framework: Policy files may redirect older versions of an assembly to newer ones. If the backwards compatible types are otherwise identical, references to types in the older assembly will be redirected to the newer one. This means that all references to an assembly use the newer version of the assembly regardless of which assembly was referenced by the program at build time. At execution time, down-versioning scenarios are converted to up-versioning by redirecting references to the latest version of an assembly. If the policy files redirect an older version of an assembly to a newer one, and if a .NET program references an older version of an assembly and tries to load a newer version of the assembly to satisfy the reference, the program will not fail because the reference is redirected to the newer version to begin with.
- .NET (6+): If a .NET program references an older version of an assembly, it can load a newer one and use only the older features, assuming that they are available. .NET (6+) handles this under the covers. However, if a .NET program references a newer version of an assembly and tries to load an older version of the assembly to satisfy the reference, the program will always fail. .NET (6+) doesn't have policy files or any equivalent.
- The most important consequence of this difference for VISA is that up-versioning is not supported in .NET (6+). This limits the flexibility of .NET (6+) versioning, and requires more customer

[1] In theory, massive changes to the IVI APIs might lead to deletions in the VISA.NET API. This would likely be interpreted as a completely new set of APIs rather than new versions of existing APIs. In any case, we do not anticipate changes on this scale, and have not made any such changes in the past.

intervention to handle cases where up-versioing is an issue.

19.3. Maintaining Backwards Compatibility

This section describes the practices needed to maintain backwards compatibility. The versioning style described in this section does not cover all of the possible ways in which the VISA.NET Shared Components could change from version to version, but it does describe most of the situations that would impact VISA.NET. In particular, it discuses the kinds of changes that are allowed for enumerations, interfaces, and classes, and calls out changes that break backwards compatibility.

19.3.1. Naming New Versions of .NET Types

Each .NET type declared in an VISA.NET Shared Components assembly shall have a base name that is version independent. The first version shall use this name without modification. For each subsequent version, the base name shall have an integer appended, starting with "2" and incrementing by 1.

For example, the first version of the Ivi.Visa GPIB session interface is named IGpibSession. The second published version of this interface would be named IGpibSession2, the third would be named IGpibSession3, and so on.

19.3.2. Versioning Enumerations

Enumerations shall not be deleted or renamed. New enumerations may be added.

Enumeration members shall not be deleted or renamed. The numeric value of an existing enumeration member shall not be changed, since it is the same as deleting the member with the old value and adding the member with the new value.

If an enumeration member must be deleted or renamed, or existing numeric values must be changed, a new enumeration shall be created. The new enumeration shall be named as described in section 0, *This section describes the practices needed* to maintain backwards compatibility. The versioning style described in this section does not cover all of the possible ways in which the VISA.NET Shared Components could change from version to version, but it does describe most of the situations that would impact VISA.NET. In particular, it discuses the kinds of changes that are allowed for enumerations, interfaces, and classes, and calls out changes that break backwards compatibility.

Naming New Versions of .NET Types. Since the intent is to allow users to migrate to the new enumeration with a minimum of change, enumeration members that are common to both the old and new versions of the enumeration shall have the same spelling and numeric values.

Enumeration members may be added if they are added in a way that does not cause the value of any existing members to change. For enumerations where numeric values are not specified, this means that new members shall only be added to the end of the enumeration.

EXAMPLE

If an enumeration named "TriggerLines" includes a trigger line called "TriggerLine0" that is no longer needed, the following versioning strategy is used:

- The "TriggerLines" enumeration is not modified.
- A new enumeration is created, named "TriggerLines2", that includes all of the old members except for "TriggerLine0".
- The new members match the old members in spelling and value. If the "TriggerLines" enumeration uses default values, values may need to be specified for "TriggerLines2" if the removal of the "TriggerLine0" member leaves a gap in the values.

19.3.3. Versioning Interfaces

Interfaces shall not be deleted or renamed. New interfaces may be added.

Interface members shall not be added or deleted, and the signatures of existing members shall not be changed in any way, including:

- The return type of an existing member shall not be changed.
- Parameters shall not be added or deleted to any member, and
- Parameter names and types shall not be changed.

If an interface member must be added, deleted, or changed in some way, a new interface shall be created. The new interface shall be named as described in section 0, *This section describes the practices needed* to maintain backwards compatibility. The versioning style described in this section does not cover all of the possible ways in which the VISA.NET Shared Components could change from version to version, but it does describe most of the situations that would impact VISA.NET. In particular, it discuses the kinds of changes that are allowed for enumerations, interfaces, and classes, and calls out changes that break backwards compatibility.

Naming New Versions of .NET Types. Since the intent is to allow users to migrate to the new interface with a minimum of change, interface members that are common to both the old and new versions of the interface shall have the same signatures.

If a new interface is created to version an older interface, it shall be created in one of two ways:

- The new interface is cloned from the older interface, and then modified within the constraints listed above. This technique will always work.
- The new interface derives from the older interface. However, derivation has pitfalls - interface reference properties may need to return references to newer interfaces, for example, and members from the derived interface may not be deleted. To accommodate these situations, the following process is followed when deriving a new interface from an older one:
    - New members are added to the new interface.
    - Where a new method or property that matches an older method or property except for the return type, the new method or property uses the "new" modifier to hide the older one. (This addresses the issue with interface reference properties.)
    - Obsolete members are tagged with the "Obsolete" attribute in the older interface. The "Obsolete" attribute is constructed so that trying to build code that uses the member generates a build warning or error (at the discretion of the VISA WG). Note that the member is still available and does not generate a runtime error for an executable built against the older version of the interface.

EXAMPLE

For example assume an interface named "IMessageBasedSession". "IMessageBasedSession" is missing a property named "AvailableBytes", and the "ReadStatusByte" method is missing a "timeout" parameter. In addition, an interface reference property called "FormattedIO" is modified to return a reference to the "IMessageBasedFormattedIO2" interface instead of the "IMessageBasedFormattedIO" interface. The following versioning strategy is used:

- The "IMessageBasedSession" interface is not modified.
- A new interface is created, named "IMessageBasedSession2", that includes the following members
    - All of the old members with the same signatures. The old "ReadStatusByte" overload is omitted.
    - A new overload of the "ReadStatusByte" method with the "timeoutMilliseconds" parameter.
    - The new "AvailableBytes" property.
    - The modified "FormattedIO" property.
- Where new members match the old members, the signatures also match.

EXAMPLE - CLONING CODE

```
public interface IMessageBasedSession : IVisaSession
{
   public Int16 ReadStatusByte ();
```

```
   // Other methods …
   public IMessageBasedFormattedIO FormattedIO { get; }
   // Other properties …
}

public interface IMessageBasedSession2 : IVisaSession
{
   public Int16 ReadStatusByte (Int32 timeoutMilliseconds);
   // Other methods …
   public IMessageBasedFormattedIO2 FormattedIO { get; }
   public Int32 AvailableBytes { get; }
   // Other properties …
}
```

EXAMPLE - DERIVATION CODE

```
public interface IMessageBasedSession : IVisaSession
{
   [Obsolete, false]  // This generates an warning on build.
   public Int16 ReadStatusByte ();
   // Other methods …
   public IMessageBasedFormattedIO FormattedIO { get; }
   // Other properties …
}

public interface IMessageBasedSession2: IMessageBasedSession
{
   public Int16 ReadStatusByte (Int32 timeoutMilliseconds);
        // Use new to hide the old Display property.
   new public IMessageBasedFormattedIO2 FormattedIO { get; }
   public Int32 AvailableBytes { get; }
}
```

19.3.4. Versioning Classes

Classes shall not be deleted or renamed. New classes may be added.

Class members shall not be deleted, and the signatures of existing members shall not be changed in any way, including:

- The return type of an existing member shall not be changed.
- Parameters shall not be added or deleted.
- Parameter names and types shall not be changed.
- For members derived from interfaces, implementation shall not be changed from explicit to implicit or vice versa.

If a class member must be deleted, or changed in some way, a new class shall be created. The new class shall be named as described in section 0, *This section describes the practices needed* to maintain backwards compatibility. The versioning style described in this section does not cover all of the possible ways in which the VISA.NET Shared Components could change from version to version, but it does describe most of the situations that would impact VISA.NET. In particular, it discuses the kinds of changes that are allowed for enumerations, interfaces, and classes, and calls out changes that break backwards compatibility.

Naming New Versions of .NET Types. Since the intent is to allow users to migrate to the new class with a minimum of change, class members that are common to both the old and new versions of the class shall have the same signatures.

New class members (including overloads) may be added to existing classes.

In general, the range of behavioral changes that don't affect the class API is fairly broad, and the decision to implement a new class or not in response to a particular behavioral change is left to the discretion of the VISA Working Group.

The only significant difference between versioning interfaces and classes is the way that new members are treated. Therefore, the techniques used to version classes are nearly the same as those used to version interfaces, with the exception that if the only change to a class is to add new members, the new members may be added to the existing class.

Exceptions are just a specialization of a class, and are versioned like classes.

19.3.5. Changes and Version Numbers

All API changes result in a change to the major or minor version number of the assembly. The decision is left to the discretion of the VISA Working Group.

Behavioral changes result in a change to the major, minor, or build version number of the assembly. The decision is left to the discretion of the VISA Working Group.

XML comments may be changed freely, and result in a change to the build version number of the assembly (if there are no other changes).

The VISA.NET Shared Components are delivered in a single assembly for ease of use.

19.4. How Versioning Works For .NET (6+)

The .NET (6+) runtime and the GRM work together to load the best possible version of the vendor VISA.NET assembly. The following description assumes that newer versions of the assembly are backwards compatible with older versions with the same major IVI VISA.NET version as described above.

19.4.1. .NET (6+) Versioning Techniques

Although .NET (6+) frequently uses application local versions of assemblies to manage versions, this approach is not sufficient to support VISA.NET. In particular,

- An application may choose to directly reference more than one vendor VISA.NET assembly. In this case NuGet is responsible for choosing the appropriate IVI VISA.NET assembly for the application. Most likely this application does not use the GRM to dynamically load vendor VISA.NETs.
- An application may choose to only directly reference the IVI VISA assembly by using the GRM. In this case the GRM dynamically loads the vendor VISAs.

  A VISA.NET application or library might dynamically load more than one vendor VISA.NET assembly. Each of those assemblies references a IVI VISA.NET assembly, but there is no guarantee that they would both reference the same version. In this case, the runtime could not decide what to load, since both libraries are dynamically loaded. The GRM will do it's best to load a compatible version. Once loaded, the runtime can satisfy older API references from the newer assembly, assuming that backwards compatibility rules have been followed.

In some cases, application developers will have to upgrade their application to directly reference the latest version of the IVI VISA.NET assembly.

19.4.2. Maintaining Software Configurations

Some programs that use VISA.NET are rigorously qualified with a given software configuration, and once qualified, are expected to build and run against that exact configuration.

Application local deployment in .NET (6+) is designed just to support this kind of scenario. If all the software is installed with an application, including all the vendor VISA.NET assemblies that can be dynamically loaded, then the application can be copied to new locations and expected to run as it did on the original system, as long as the rest of the system configuration that might impact the application is qualified also.

Note that most VISA.NET implementations delegate most, if not all, of their functionality to the vendor's corresponding VISA-C implementation, and vendor VISA-C implementations are global, not local. When rigorously qualifying an application, the versions of any installed vendor VISA-C implementations should be considered.

19.4.3. VISA.NET Shared Components NuGet Package

The VISA.NET NuGet package version major/minor version shall be the same as the VISA.NET assembly major/minor version. The package build number is created by the build and independent of the major/minor version requirements.

Note that the Conflict Manager C DLL is installed by the VISA Shared Components (which are a pre-requisite for the VISA.NET Shared Components), and so there is no connection between the version of that DLL and the VISA.NET assembly or shared components installer.

19.5. How Versioning Works For .NET Framework

For .NET Framework assemblies, side-by-side installation allows multiple versions of a .NET assembly to be installed side-by-side (e.g. at the same time) and publisher policy files direct references from older versions of an assembly to a newer version of the assembly, The following description assumes that newer versions of the assembly are backwards compatible with older versions as described above.[1]

19.5.1. Versioning with Policy Files

In order to meet versioning objectives, the VISA.NET Framework shared components provide publisher policy files to redirect references from older versions of the assembly to the newer version.[2] (When the term "policy file" is used in this document without qualification, it refers to publisher policy files.)

Using side-by-side versioning without policy files for shared component versioning violates this principle.

- User code that references shared components data types would be exposed to different versions of the same shared component data types.
- An application that used multiple VISA.NET implementations would not be able to simultaneously reference or use implementations that referenced different versions of the VISA.NET Shared Components without taking measures that violate the versioning principle (such as isolating the calls to drivers that use different versions of the shared components into separate DLLs).

Using publisher policy files implies that the assemblies continue to provide the older versions of the interfaces along with new ones. If assemblies do not continue to support older versions, the versioning principle is also violated.

- If an application uses Vendor A's implementation built with an older version of the shared components and Vendor B's implementation built with a newer version of the shared components that revises an Interface that Vendor A's implementation uses, Vendor A's implementation would break when the shared components are loaded, because the older version of the interface would not be available.

[1] Side-by-side versioning without policy files is only absolutely required when the target .NET Framework of the assemblies change, and the change results in using a version of the .NET Common Language Runtime (CLR) that is not compatible with the previous version. The IVI Foundation will only make changes for this reason when the current target .NET Framework version becomes unsupported. Massive changes to the IVI APIs could also trigger such a change, but this would likely be interpreted as a completely new set of APIs, and we do not anticipate changes on this scale.

[2] The following are relevant observations about .NET, and are not within the control of the IVI Foundation:

- .NET requires exactly one publisher policy file for each old major/minor version when using policy files to version up. The old major and minor version numbers are part of the policy file name.
- Adding methods or properties to an interface will break components built against the old interface, because the new method/property will not be implemented by the component.

19.5.2. Maintaining Software Configurations

Some programs that use VISA.NET are rigorously qualified with a given software configuration, and once qualified, are expected to build and run against that exact configuration. Installing publisher policy files that redirect assembly references to new versions of an assembly might violate this expectation.

To accommodate users who need to strictly control their software configuration, multiple versions of the VISA.NET Shared Components can be installed side-by-side with later versions. In these cases, references to older versions of the VISA.NET assembly will, by default, be redirected to the latest installed version using publisher policy files. The default behavior may be overridden by using application or machine policy files. VISA.NET vendors are responsible for providing instructions for end users who might wish to do this.

Ordinarily these versions will not be used at run time, since any run time reference to them will "policy up" to the newest installed version. However, if a client wishes to continue using an older version of an assembly, an application configuration file (probably most common) or machine configuration file may be created that maintains references to the older version. VISA.NET vendors should be prepared to support customers who need to use older versions of an assembly.

When developing code, references to the specific version required for the application can be added to a project and will be used consistently after that point for editing and building the project, as long as that specific version is installed.

19.5.3. Versioning for Policy Files

Any changes to an assembly require that the assembly have a new version number and that the policy file(s) be updated to refer to the new version of the assembly.

19.5.4. VISA.NET Shared Components Installer

The VISA.NET installer version major/minor version shall be the same as the VISA.NET assembly major/minor version. In some cases where the only changes are to the installer, the installer build number may be greater than the assembly build numbers.

Note that the Conflict Manager DLL is installed by the VISA Shared Components (which are a pre-requisite for the VISA.NET Shared Components), and so there is no connection between the version of that DLL and the VISA.NET assembly or shared components installer.

19.6. VISA.NET Implementations

It is recommended that VISA.NET implementations use the versioning style for the VISA.NET Shared Components, except that one of the restrictions on interface versioning may be loosened. In particular, interface members may be added to an interface without creating a new numbered version of the interface (and retaining the original interface) if the vendor does not support any other interface implementations outside of the assembly.